AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE
Wydział Fizyki i Informatyki Stosowanej

Praca magisterska

Łukasz Pawlik

kierunek studiów: **Informatyka Stosowana**

Opracowanie systemu szybkiego przesyłania danych z wykorzystaniem standardu USB

Promotor: **dr inż. Bartosz Mindur**

Kraków, 2015

Oświadczam, świadoma odpowiedzialności karnej za poświadczenie nieprawdy, że niniejszą pracę dyplomową wykonałam osobiście i samodzielnie i nie korzystałam ze źródeł innych niż wymienione w pracy.

.....

(czytelny podpis)

Merytoryczna ocena pracy przez Opiekuna:

Ocena końcowa pracy przez Opiekuna:

Data: Podpis: .....

Merytoryczna ocena pracy przez Recenzenta:

Ocena końcowa pracy przez Recenzenta:

Data: Podpis: .....

Spis treści

Spis elementów implementacji

I. Wstęp

W dobie dzisiejszych możliwości rosną również wymagania i oczekiwania. Dotyczy to prawie każdego aspektu dzisiejszej egzystencji począwszy od spraw osobistych poprzez finansowe aż po możliwości uzyskiwania danych. Rozwój technologi, zwłaszcza w zakresie elektroniki, w przeciągu minionego ćwierćwiecza stał się imponujący. Udało się doprowadzić różne skomplikowane układy scalone do rozmiarów wielkości pudełka zapałek, których złożoność obliczeniowa przekracza najśmielsze oczekiwania z przed kilku lat. Dla współczesnego człowieka problemem okazuje się czas, zawsze jest go za mało w praktycznie każdej sferze życia. Jest on również istotny w rozwoju technologi oraz uzyskiwaniu konkretnych danych pomiarowych.

Dzisiejsze urządzenia pomiarowe laboratoryjne jak i medyczne (oraz inne) mają za zadanie zbieranie i przesyłanie danych celem późniejszej wizualizacji lub analizy za pomocą dedykowanego oprogramowania. Dane te z reguły są wysyłane w określonym formacie dla danego urządzenia a co za tym idzie rozmiar wysyłanych danych musi być większy niż w wypadku danych wysyłanych bajt po bajcie.

Człowiek odkąd powstał pierwszy komputer rozpoczął walkę z problemem komunikacji między urządzeniami. Zostało opracowanych wiele standardów komunikacji, niektóre projekty zostały skazane na zapomnienie, inne odniosły sukces i zdominowały urządzenia. Takim standardem jest z pewnością USB, czyli Uniwersalna Magistrala Szeregowa (dokładny opis wraz z rysem historycznym w rozdziale II.2). Na dzień dzisiejszy większość urządzenia dla których konieczna jest komunikacja z komputerem osobistym (wyłączając sieci Ethernetowe [1] oraz WiFi[2]) posiada interfejs USB.

Celem niniejszej pracy było opracowanie szybkiego przesyłania danych używając standardu USB. Przez szybkie rozumiana jest tutaj prędkość nie mniejsza niż 140Mbit/s (17,5 MB/s) [3].

Pierwszym zadaniem, które musiało zostać wykonane przed rozpoczęciem realizacji implementacji oprogramowanie było wybranie hardwaru, które miało pełnić role urządzenia testowego (nie jest możliwe wykonanie testów przesyłu danych po interfejsie USB bez urządzenia odbierającego dane). Dokładny opis wybranego urządzenie został umieszczony w rozdziale V. Kolejnym zadaniem był odpowiedni wybór biblioteki spełniającej założenia projektu. Istnieje wiele implementacji bibliotek udostępniających API [4] dla interfejsu USB. W rozdziale III szczegółowo zostały opisane dwie najczęściej używane z których pierwsza używana jest zarówno pod systemem Windows jak i pod systemem Unix natomiast użycie dru-

[1] Ethernet - standard sieci lokalnych

[2] WiFi - technologia sieci bezprzewodowych

[3] 1 B = 8 bit

[4] API (ang. Application Programming Interface) - zestaw funkcji oraz innych narzędzi umożliwiających wykorzystanie możliwości danej biblioteki

giej z nich możliwe jest tylko pod systemami Windows. Dodatkowo zostało przedstawione krótkie uzasadnienie wyboru.

Kolejnym etapem podczas projektowania oprogramowania było dokładne zaplanowanie jak przebiegać będą testy. Istotnym elementem było to iż wykonanie ich bez urządzenia zewnętrznego było niemożliwe. Wtedy też należało opracować wejściowy jak i zwracany format danych dla każdego testu. Po opracowaniu podstawowych wymagań i testowych restrykcji dla aplikacji możliwe było przystąpienie do pisania aplikacji. Opis aplikacji wraz z implementacją znajduje się w rozdziale VI.

Ostatnim i zarazem najważniejszym elementem pracy jest zestawienie otrzymanych wyników wraz z oczekiwaniami. Całość została opisana w rozdziale VII i zawiera konkretne wnioski na temat realizacji projektu oraz ogólnych możliwości interfejsu USB. W rozdziale zawarte jest dodatkowo zestawienie i interpretacja otrzymanych wyników w zakresie synchronicznego oraz asynchronicznego przesyłania danych za pomocą USB. Została w nim zaprezentowana dodatkowa symulacja, aby sprawdzić zachowanie interfejsu w mniej dostępnym środowisku.

II. Motywacje projektu oraz podstawowe założenia

Podstawową motywacją rozpoczęcia pracy nad projektem była chęć zdobycia wiedzy w zakresie komunikacji opartej na interfejsach szeregowych. Jest to wiedza która może zostać wykorzystana w przyszłości a tego typu projekty z pewnością prowadzą do samodoskonalenia. Ważnym elementem było zaczerpniecie jak największej wiedzy na temat interfesów szeregowych oraz samego USB które jest przedmiotem niniejszej pracy.

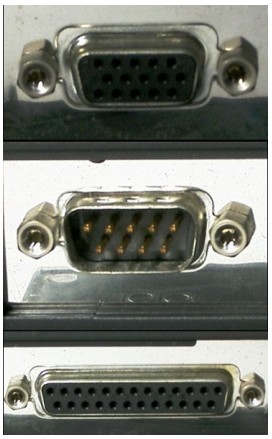
II.1. Czym są interfejsy szeregowe i do czego służą

Porty szeregowe w kontekście informatycznym są to fizyczne interfejsy umożliwiające przesłanie danych z lub do urządzenia w jednostce czasu jeden za drugim. Cechy interfejsów szeregowych można opisać za pomocą trzech prostych punktów.

- prostota
- wydajność
- trudność implementacji

Do najprostszych interfejsów szeregowych zaliczamy z pewnością interfejsy takie jak One-Wire oraz RS232C (kabel przedstawiony na rysunku II.1).

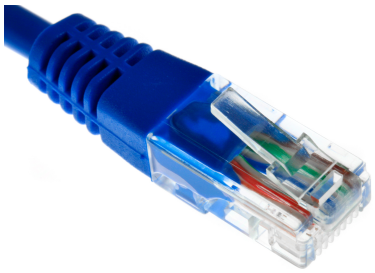
Rys. II.1. *RS232C*



Rys. II.2. *RS232family*

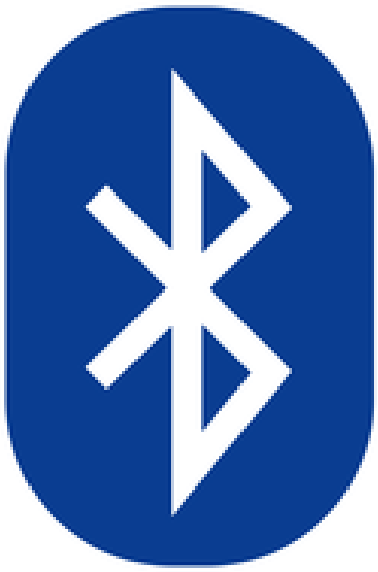
Złącza typu RS232C można jeszcze spotkać jako interfejs komunikacyjny z drukarkami starego typu, niemniej zostało on wyparty przez nowsze interfejsy (takie jak n.p. USB).

Interfejsami najbardziej wydajnymi są USB oraz Ethernet. Są to stosunkowo nowo opracowane (w porównaniu do RS232) i nadal rozwijane interfejsy. Dokładny opis standardu USB znajduje się w rozdziale II.2. Ethernet jest to ustandaryzowany sposób komunikacji umożliwiający tworzenie komputerowych sieci lokalnych. Początki sięgają roku 1976. Opracowany został w firmie Xerox.



Rys. II.3. *EthernetCable*

Jeśli chodzi o trudność implementacji to najtrudniejszym do zaimplementowania protokołem/interfejsem przesyłania danych jest Bluetooth. Służy on do bezprzewodowej komunikacji na bardzo krótkim dystansie.



Rys. II.4. *bluetoothLogo*

Dość spotykanym przypadkiem użycia komunikacji bluetooth są zestawy głośno mówiące w samochodach oraz różnego rodzaju urządzenia takie jak słuchawki, głośniki oraz adaptory (np. w zestawie do bezprzewodowej myszki komputerowej dołączany jest adapter bluetooth podpinany pod wejście USB w komputerze).

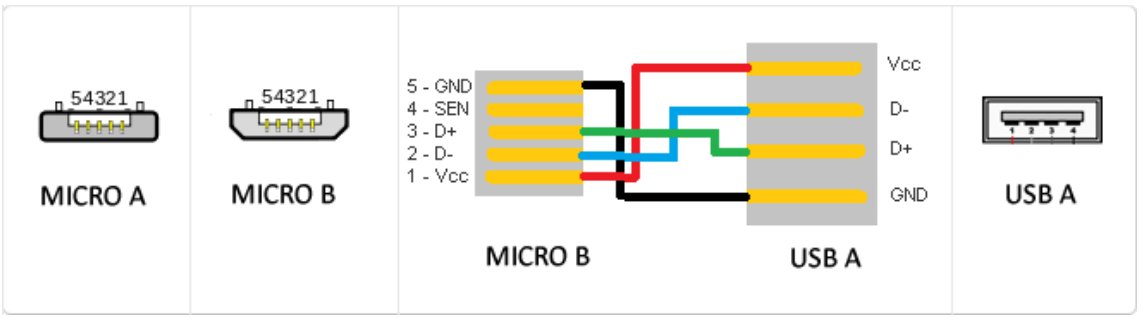
Podsumowując, interfejsy szeregowe umożliwiają komunikację miedzy urządzeniami z których jeden najczęściej jest komputerem osobistym. Implementacja każdego interfejsu jest na swój sposób złożona. Kategoryzacja jest możliwa na podstawie różnych wartości. W tym rozdziale zostały podzielone według prostoty interfejsu (w sensie architektury), wydajności oraz trudności implementacji.

II.2. Czym jest USB

Uniwersalna Magistrala Szeregowa jest to standard opracowany w latach 90. XX w. definiujący jakie kable, złącza oraz protokoły mają być używane podczas połączenia, komunikacji oraz definiuje sposób zasilania pomiędzy komputerem i urządzeniem elektronicznym. USB zostało zaprojektowane aby ułatwić połączenia standardowych elektronicznych urządzeń takich jak klawiatury, myszki, drukarki, aparaty cyfrowe, dyski przenośne do komputerów osobistych. Wszystkie te urządzenia są dodatkowo zasilane również za pomocą tego portu. Z czasem stało się to wspólne również dla innych urządzeń takich jak smartphony, palmtopy oraz konsole wideo.[3, 6, 7]

USB szybko zastąpiło porty szeregowe oraz równoległe podobnie jak inne urządzenia zasilające elektroniczne urządzenia. Istnieją trzy podstawowe rodzaje złączy USB dla których kryterium podziału stanowi wielkość. Najstarszy rozmiar (używany np. w pendrive'ach) występuje w standardach USB1.1, USB2.0, USB3.0, mini-USB (początkowo tylko dla złącza typu B, jak w wypadku wielu aparatów cyfrowych) oraz mikro-USB występuje również w trzech wariantach dla USB1.1, USB2.0, USB3.0 (dla przykładu używany w nowych telefonach komórkowych).

W przeciwieństwie do innych kabli do przesyłu danych (np. Ethernet, HDMI) każdy koniec kabla zakończony jest innym typem złącza (typem A lub typem B). Tylko złącze typu A dostarcza zasilanie. Zostało to zaprojektowane w taki sposób aby uniknąć elektrycznych przeciążeń a co za tym idzie uszkodzeniu urządzeniu. Istnieją również kable ze złączami typu A na obu końcach, ale nie należą do popularnych (i należy postępować z nimi ostrożnie). Kable USB maja zazwyczaj złącze typu A z jednej strony oraz złącze typu B z drugiej oraz wejście w komputerze lub urządzeniu elektronicznym. w przyjętej praktyce złącze typu A jest zazwyczaj największej (z możliwych wielkości), natomiast B w zależności od potrzeb użycia kabla (full, mini, micro). [3, 6, 7]



Rys. II.5. *Połączenie przewodów w micro-USB*

USB zapoczątkowało w 1994 siedem firm: Compaq, DEC, IBM, Intel, Microsoft, NEC, Nortel. Celem było uproszczenie podłączenia zewnętrznych urządzeń do komputera zastępując stare złącza w płytach głównych wprowadzając rozwiązania na problemy znalezione

w starych oraz upraszczając software. Pierwszy układ scalony wspierający USB został wyprodukowany przez Intel 1995r.

Pierwsza oficjalna wersja standardu USB została wydana w styczniu 1996r. USB1.0 charakteryzowała prędkość 1,5 Mbit/s (Low Speed) oraz 12 Mbit/s (Full Speed). Nie pozwalał jednak na używanie przedłużaczy kabli, wynikało to z limitów zasilania. Powstało kilka wypuszczono na rynek na chwile przed wydaniem standardu USB1.1 w sierpniu 1998r. W USB1.1 poprawiono kilka błędów znalezionych w USB1.0 i był to pierwszy standard, który został oficjalnie zaimplementowany w standardowych komputerach osobistych. [3]

USB2.0 zostało wydane w kwietniu 2000r. udostępniając maksymalny przesył sygnału rzędu 480 Mbit/s (60MB/s) nazwany High Speed (USB1.x za pomocą Full Speed umożliwiał przesył rzędu 12Mbit/s). Biorąc pod uwagę zależności dostępu do magistrali przepustowość High Speed ogranicza się do 280 Mbit/s (35 MB/s). Przyszłe modyfikacje do specyfikacji USB zostały zaimplementowane przez "Engineering Change Notitices" (ECN). Najważniejsze z ECNów zostały dołączone do specyfikacji USB2.0 dostępnej na stronie internetowej USB.org. Przykłady ECNow: Złącze Mini-A oraz Mini-B: wydane w październiku 2000r.[6]

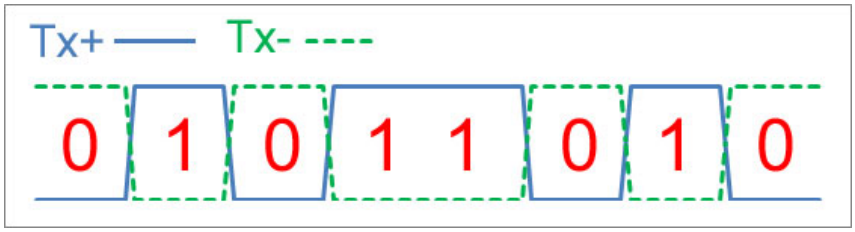
Standard USB3.0 został wydany w listopadzie 2008r. definiujący zupełnie nowy tryb "SuperSpeed". Port USB zwyczajowo jest w kolorze niebieskim i kompatybilny z urządzeniami USB2.0 oraz kablami. Dokładnie 17 listopada 2008r. ogłoszono iż specyfikacja dla wersji 3.0 została całkowicie ukończona i została zaakceptowana przez "USB Implementers Forum" (USB-IF), czyli głównej instytucji zajmującej się specyfikacjami standardu USB. To pozwoliło na szybkie udostępnienie standardu deweloperom. Nowa magistrala "SuperSpeed" dostarcza czwarty typ transferu z możliwością przesyłania sygnału z prędkością 5GBit/s, ale poprzez użycie kodowania 8b/10b przepustowość wynosi 4Gbit/s. Specyfikacja uznaje za zasadne osiągniecie prędkości w okolicach 3,2 Gbit/s (400 MB/s) co w założeniach powinno się zwiększać wraz z rozwijaniem hardwaru. Komunikacja odbywa się w obu kierunkach dla SuperSpeed (kierunek nie jest naprzemienny i nie jest kontrolowany przez hosta, jak to ma miejsce do wersji USB2.0). Podobnie jak w poprzednich wersjach standardu, porty USB3.0 działają dwóch wariantach zasilania: niskiego poboru mocy (low-power: 150mA) oraz wysokiego poboru mocy (high-power: 900mA). Zapewniając odpowiedni jednocześnie pozwalają na przesył danych z prędkością SuperSpeed. Została dodatkowo zdefiniowana specyfikacja zasilania (w wersji 1.2 wydana w w grudniu 2010r.) która zwiększała dopuszczalny pobór mocy do 1,5A, ale nie pozwala na współbieżne przesyłanie danych. Specyfikacja wymaga aby fizyczne porty same w sobie były wstanie obsłużyć 5A, ale ogranicza pobór do 1,5 A. [7]

W styczniu 2013r. w prasie pojawiły się informacje o planach udoskonalenia standardu USB3.0 do 10Gbit/s. Zakończyło się to stworzeniem nowej wersji standardu - USB3.1. Wersja ta została wydana 31 lipca 2013r. wprowadzając szybszy typ przesyłania danych zwany "SuperSpeed USB 10 Gbit/s". Zaprezentowano również nowe logo stylizowane na zasadzie

"Superspeed+". Standard USB3.1 zwiększył szybkość przesyłu sygnału do 10Gbit/s. Udało się też zredukować obciążenie łącza do 3% dzięki zmianie kodowania na 128b/132b.
Przy pierwszych testach prędkośi USB3.1 udało się uzyskać prędkość 7,2Gbit/s.
Standard USB3.1 jest wstecznie kompatybilny ze standardem USB3.0 oraz USB2.0. [7]

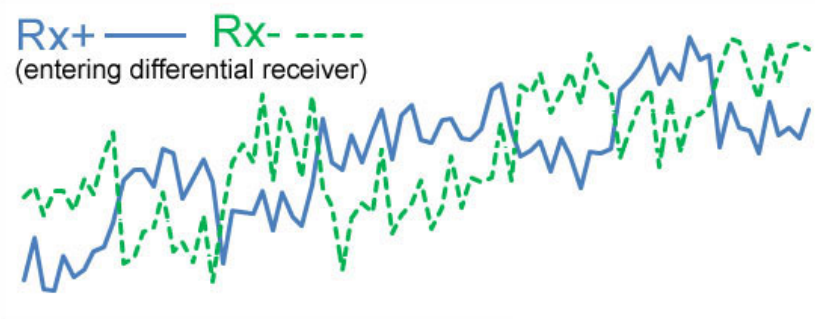
Poszczególne rodzaje kodowania dla standardów USB3.0 oraz USB3.1 to kodowanie 8bitów na 10 bitów oraz 128 bitów na 132 bity. Jeśli chodzi o przesyłanie danych to do momentu wprowadzenia Differential signaling (polega na wysłaniu dwóch przeciwnych sobie sygnałów TX+ oraz TX- do zewnętrznego odbiornika, który aby zniwelować zaszumienie odejmuje wartość RX- od obydwu kanałów) przesyłanie danych na długie odległości lub wysyłanie danych z większą prędkością było awykonalne. Przykład wraz z kolejnymi etapami widoczny na rys II.6, II.7, II.8.

W czasach kiedy elektronika miała swoje początki zarówno nadawca jak i odbiorca współdzielili to samo uziemienie (były częścią jednego środowiska), więc nie było problemów aby zmierzyć napięcie sygnału wejściowego. Więc jeżeli założenie było iż 0V reprezentuje wartość bitu równą ’0’ a wartość +5V reprezentuje wartość bitu równą ’1’ oraz zegar był również wspólny dla nadajnika oraz odbiorcy, odbiorca świetnie potrafił odczytać nadawany sygnał (np. rys. II.6).

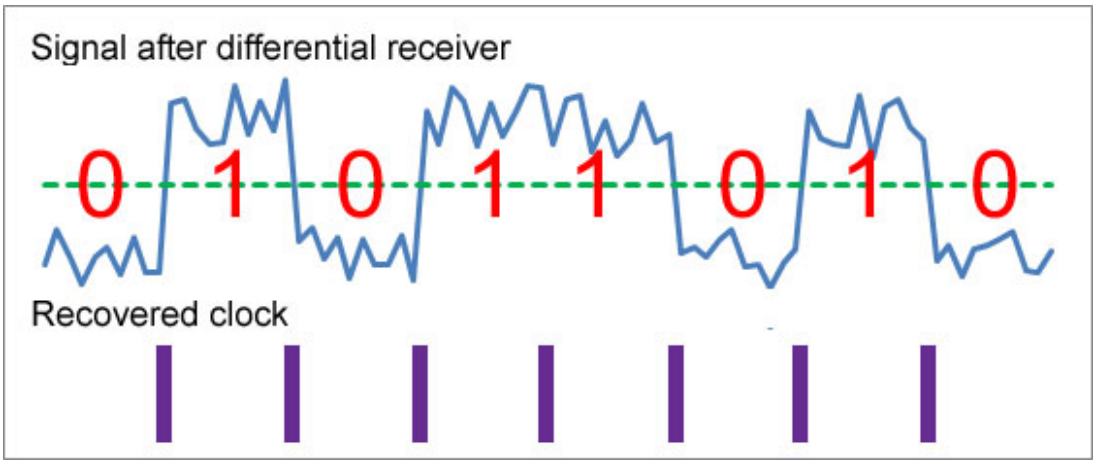


Rys. II.6. *cleanSignal.*

Problem pojawia się w momencie kiedy odbiornik znajduje się poza środowiskiem nadawcy (nadawca i odbiorca nie współdzielą uziemienia, zegara, etc.). Rys. II.7 przedstawia sygnał odebrany po stronie odbiornika (jest to ten sam sygnał co na rys. II.6). Przez brak współdzielenia uziemienia oraz zegara przez nadajnik oraz odbiornik sygnał stał się nieczytelny po stronie odbiornika.



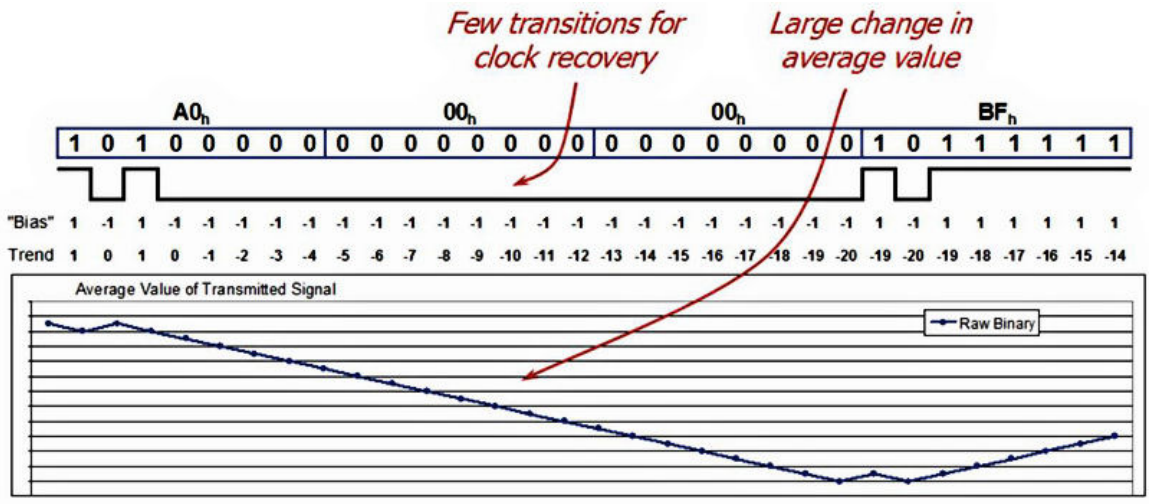
Rys. II.7. *receivedNoiseSignal.*



Rys. II.8. *receivedCleanedSignal.*

Rozwiązaniem problemu przesyłu danych między dwoma zewnętrznymi okazało się zaprojektowanie differential receiver'a, czyli odbiornika odbierającego dwa sygnały RX+ oraz RX-. Poziom sygnału jest mierzony na podstawie tych dwóch sygnałów. Skrętka[1] gwarantuje, że przesunięcie napięcia (rys. II.7) dla obu kanałów jest takie samo a więc różnica pomiędzy jest stała i jest możliwa do odczytu przez odbiornik (rys. II.8). Dodatkowo odbiornik jest w stanie odczytać sygnał zegara na podstawie szybkości zmian sygnału (rys. II.8).

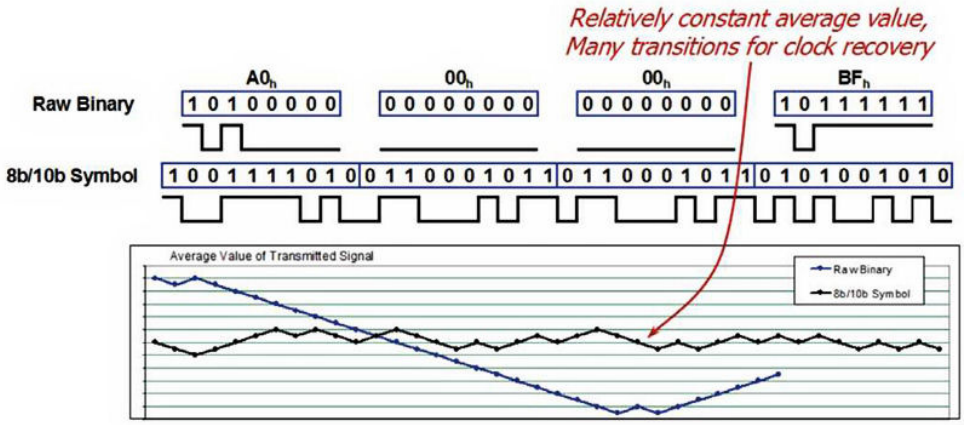
W przypadku wysyłania większej ilości danych ukazuje się kolejny problem. Jeżeli ilość tych samych wartości bitu (zer lub jedynek) występujących po sobie jest zbyt duża odbiornik może nie być wstanie odczytać sygnału zegara, lub zrobić to błędnie. Rys. II.9 zawiera przykładowy zestaw wysyłania większej ilości zer, jak widzimy w przykładzie, w drugim i trzecim pakiecie danych wysyłane są same zera a ilość przeskoków sygnału (zmian 0->1 lub 1->0) jest znikoma i obliczenie sygnału zegara może być problematyczne dla odbiornika.



Rys. II.9. *withoutEncoding*

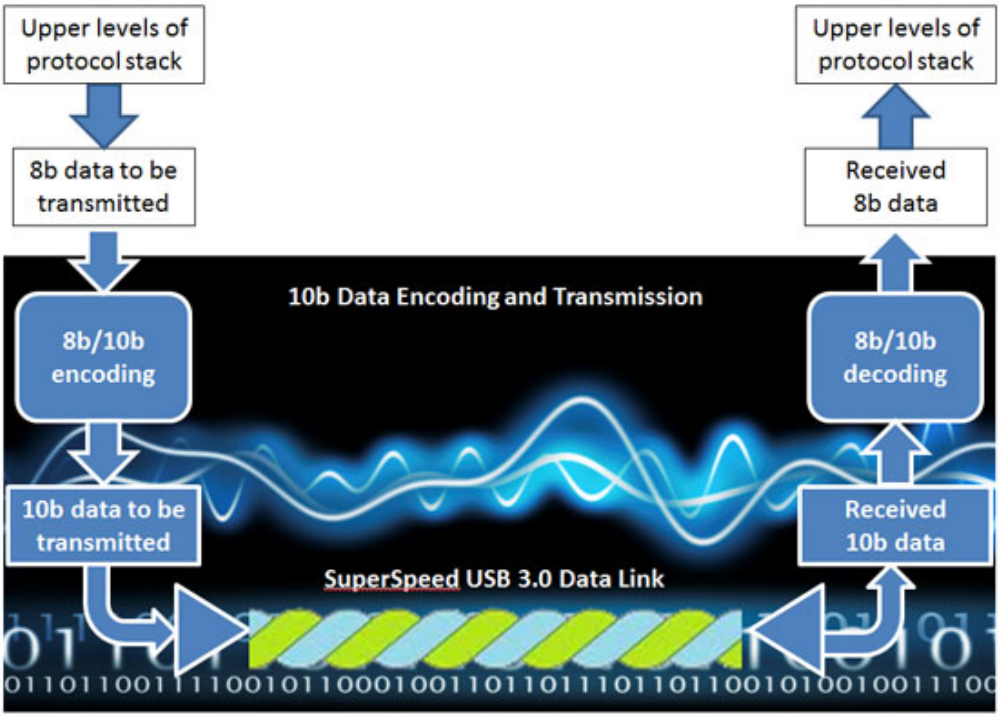
[1]Skrętka - w kontekście pary skręconych przewodów

Rozwiązaniem tego problemu jest kodowanie 8b/10b, dzięki któremu ilość zmian sygnału z zera na jeden oscyluje w okolicach 50% w stosunku do zmian z jeden na zero dzięki czemu obliczenie zegara na odbiorniku nie jest problematyczne. Przykład sygnału po zastosowaniu kodowania 8b/10b jest widoczny na rys. II.10.



Rys. II.10. *withEncoding*

Rys. II.11 przedstawia działanie USB3.0 wraz z kodowaniem oraz komunikacją przez warstwę fizyczną. Najpierw dane do wysłania przekonwertowywane są na z postaci 8-bitowej na postać 10-bitową, następnie wysyłane do odbiornika, gdzie następuje konwersja odwrotna do postaci 8-bitowej.

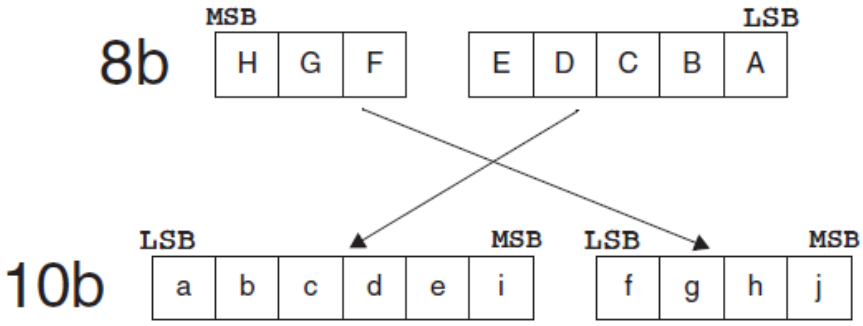


Rys. II.11. *sendingDataThroughPHYLayer*

Kodowanie 8b/10b zbudowane jest na podstawie kodowania 5b/6b oraz 3b/4b i jest ich połączeniem w odpowiedni sposób. Cały algorytm tworzenia przedstawia się następująco.

1. 8 bitów jako dane wejściowe: HGFEDCBA[2]
2. na ostatnich 5 bitach (EDCBA) wykonujemy kodowanie 5b/6b (dochodzi dodatkowy bit)
3. na pierwszych trzech bitach (HGF) wykonujemy kodowanie 3b/4b (dochodzi kolejny bit)
4. całość łączona do postaci abcdeifghj[3] tworząc rezultat

Rysunek II.12 doskonale przedstawia ten proces.



Rys. II.12. *810encoding mapping*

Dwa ostatnie bity ustala się oddzielnie dla tych dwóch kodowań, ustala się je na podstawie współczynnika RD[4], czyli różnicy pomiędzy ilością jedynek i zer w ciągu. Jak zostało wspomniane wyżej założenie całego kodowania jest takie aby ilość zer całości przesyłanych danych oscylowała w okolicach 50%, więc różnica (RD) dla jednego zestawu 10 bitów danych wynosi +-2.

Realizacja kodowania 5b6b jest nieco bardziej skomplikowana, polega to na takim dopasowaniu ’0’ i ’1’ aby różnica pomiędzy ilością ’0’ oraz ’1’ była równa +-1 (jest to tak na prawdę RD lub odległość Hamminga[5]). Wartość wyjściowa musi być unikalna. Jest to możliwe ponieważ dodawany jest dodatkowy bit (unikalność mogła by zostać zachowana bez niego, natomiast nie zostałby spełniony warunek RD = +-1).

[2]HGFEDCBA - liczba zapisana binarnie w postaci 8 bitów, gdzie H jest najbardziej znaczącym bitem a A jest najmniej znaczącym bitem

[3]abcdifhgj - 10 bitowy rezultat kodowania 8b/10b, należy zwrócić uwagę iż kolejność najbardziej znaczącego i najmniej znaczącego bitu jest odwrócona w stosunku do danych wejściowych, oraz i, j, są dodatkowymi dwoma bitami

[4]RD (ang. Running Disparity) - różnica pomiędzy ilością jedynek i zer w ciągu)

[5]odległość Hamminga - w kontekście projektu XOR dwóch zbiorów

		RD = -1	RD = +1			RD = -1	RD = +1
	EDCBA	abcdei	abcdei		EDCBA	abcdei	abcdei
D.00	00000	100111	011000	D.16	10000	011011	100100
D.01	00001	011101	100010	D.17	10001	100011	
D.02	00010	101101	010010	D.18	10010	010011	
D.03	00011	110001		D.19	10011	110010	
D.04	00100	110101	001010	D.20	10100	001011	
D.05	00101	101001		D.21	10101	101010	
D.06	00110	011001		D.22	10110	011010	
D.07	00111	111000	000111	D.23	10111	111010	000101
D.08	01000	111001	000110	D.24	11000	110011	001100
D.09	01001	100101		D.25	11001	100110	
D.10	01010	010101		D.26	11010	010110	
D.11	01011	110100		D.27	11011	110110	001001
D.12	01100	001101		D.28	11100	001110	
D.13	01101	101100		D.29	11101	101110	010001
D.14	01110	011100		D.30	11110	011110	100001
D.15	01111	010111	101000	D.31	11111	101011	010100
				K.28	11100	001111	110000

Rys. II.1. *Todo:My caption 5 na 6*

		RD = -1	RD = +1			RD = -1	RD = +1
	HGF	fghj	fghj		HGF	fghj	fghj
D.x.0	000	1011	0100	K.x.0	000	1011	0100
D.x.1	001	1001		K.x.1	001	0110	1001
D.x.2	010	0101		K.x.2	010	1010	0101
D.x.3	011	1100	0011	K.x.3	011	1100	0011
D.x.4	100	1101	0010	K.x.4	100	1101	0010
D.x.5	101	1010		K.x.5	101	0101	1010
D.x.6	110	0110		K.x.6	110	1001	0110
D.x.P7	111	1110	0001				
D.x.A7	111	0111	1000	K.x.7	111	0111	1000

Rys. II.2. *Todo:My caption 5 na 6*

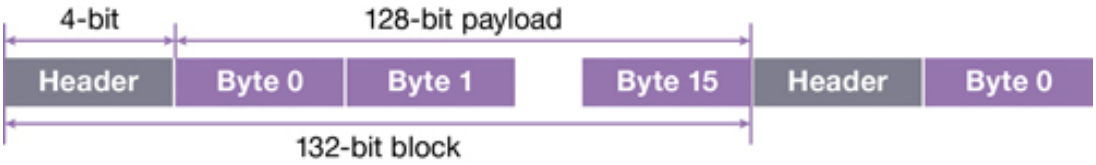
Sporą część kodów da się wyliczyć na podstawie zamiany EDCBA -> abcfe (odwrócenie bitów) oraz dodanie wartości ’0’ lub ’1’ w zależności od aktualnej wartości RD. Niestety nie

dotyczy to wszystkich elementów, tabela II.1 ukazuje wszystkie możliwe konfiguracje dla tej konwersji (5b/6b). Tabela przedstawia 32 wartości wraz z przekonwertowanymi odpowiednikami (oznaczane D.xx) oraz jedną konwersje dla instrukcji kontrolnej (oznaczona jako K.28).
Tabela II.2 przedstawia konwersje 3b/4b. Zawiera dwie specyficzne konwersje: Podstawową (ang. primary) D.x.P7 oraz Alternatywną (ang. alternative) D.x.A7. Dodatkowa alternatywna konfiguracja dla tego zestawu danych została stworzona w celu uniknięcia 5-elementowego ciągu złożonego z samych zer lub jedynek podczas łączenia z kodowaniem 5b/6b czyli tworzeniem kodu 8b/10b (a w wypadku samego podstawowego zestawu byłoby to o wiele trudniejsze).
Table II.3 przedstawia instrukcje kontrolne w postaci zakodowanej w zależności od Running Disparity (RD).

				RD = -1	RD = +1
	DEC	HEX	HGF EDCBA	abcdei fghj	abcdei fghj
K.28.0	28	1C	000 11100	"001111 0100	"110000 1011
K.28.1	60	3C	"001 11100"	"001111 1001	"110000 0110
K.28.2	92	5C	"010 11100	"001111 0101	"110000 1010
K.28.3	124	7C	"011 11100	"001111 0011	"110000 1100
K.28.4	156	9C	"100 11100	"001111 0010	"110000 1101
K.28.5	188	BC	"101 11100	"001111 1010	"110000 0101
K.28.6	220	DC	"110 11100	"001111 0110	"110000 1001
K.28.7	252	FC	"111 11100	"001111 1000	"110000 0111
K.23.7	247	F7	"111 10111	"111010 1000	"000101 0111
K.27.7	251	FB	"111 11011	"110110 1000	"001001 0111
K.29.7	253	FD	"111 11101	"101110 1000	"010001 0111
K.30.7	254	FE	"111 11110	"011110 1000	"100001 0111

Rys. II.3. *Todo:My caption Controll*

Kodowane zaimplementowane w standardzie USB3.1 (128b/132b) różni się nieco od swojego poprzednika. W kodowaniu 128b/132b pierwsze cztery bity zostały przeznaczone na header a pozostałe 128 bitów na dane. W porównaniu do kodowania 8b/10b, gdzie mieliśmy tak na prawdę utratę 20% możliwości łącza przez reprezentację każdej porcji 8 bitów - 10 bitami to tutaj mamy utratę rzędu 3%. Dzięki pierwszym czterem bitom możliwe jest określenie czy przesyłane są dane do przetworzenia czy wysyłane jest polecenie kontrolne. Dane są obudowywane w pakiety o znanym rozmiarze (zawartym w headerze) o maksymalnej wartości równej 128 bity.



Rys. II.13. *todo:128to132coding*

Podsumowując USB należy do rozbudowanych interfejsów poprzez wieloletnie udoskonalanie produktu oraz dążenie do poprawy szybkości przesyłu danych. Dzięki pracy wielu inżynierów oraz udoskonalaniu kodowania znaków aby rozwiązywać coraz to nowo napotykane problemy jest to najbardziej powszechny interfejs udostępniony do użytku publicznego.

II.3. Testy jednostkowe, komponentowe, automatyczne i wydajnościowe

Tworzenie testów jest nieodłączną częścią tworzenia oprogramowania. Istnieje wiele metodologi tworzenia testów, do najbardziej popularnych należy TDD[6] polegająca na stworzeniu przez programistę/testera testu na podstawie wymagań danej funkcjonalności zanim powstanie do niego implementacja właściwa. Metodologia ta ma swoje plusy jak i minusy. Plusem z pewnością jest weryfikacja funkcjonalności w trakcie pisania kodu produktu (widoczny jest postęp, choć sam w sobie test jeszcze nie przechodzi). Po zakończeniu funkcjonalności możliwy jest pożądany refaktoring kodu, mający na celu większą przejrzystość dla osób rozwijających projekt dalej. Dzięki zaimplementowanym wcześniej testom istnieje możliwość wstecznego sprawdzania funkcjonalności, innymi słowy czy refaktoring zbytnio nie uszkodził funkcjonalności stworzonego kodu.

Testy dzielą się na kilka grup ze względu na ich funkcjonalność. Podstawowym rodzajem są testy jednostkowe (UT[7]) polegające na testowaniu zachowania poszczególnej funkcji czy też metody. Internet szerzy się od różnego rodzaju frameworków dla UT. Do jednych z najpopularniejszych należy Google Test (jeśli chodzi o C++), który posiada bardzo łatwy do nauczenia się i przystępny framework. Za pomocą testów jednostkowych można testować wynik wyjściowy poszczególnych metod/funkcji, jak również sprawdzać ich działanie jeśli za argumenty przyjmują wskaźniki. Istnieje możliwość łatwego zbadania co dana metoda zrobiła zdanym obiektem (którego wskaźnik został podany jako argument).

Kolejnym rodzajem testów jakie powinny być przeprowadzane w miarę możliwości są testy komponentowe (SCT[8]). W tym wypadku testowanie odbywa się na zupełnie innych zasa-

[6] TDD - ang. Test Driven Development
[7] UT - Unit Tests
[8] SCT - Software Component Tests

dach i nie zawsze jest aplikowalne do każdego projektu. Testy komponentowe należy tworzyć jeżeli projekt/produkt zawiera w sobie więcej niż jeden komponent[9] oraz istnieje zaprojektowana interakcja na interfejsach zdefiniowanych miedzy nimi. Innymi słowy warunkiem koniecznym jest aby te dwa (lub więcej) komponenty musiały się ze sobą komunikować za pomocą wiadomości. Istotą komponent testów jest to, iż przygotowuje się zbliżone środowisko do docelowego (jeżeli aplikacja działa na hardwarze o określonej architekturze katalogów to należy dostarczyć zbliżoną architekturę do docelowej, często stosuje się do tego kontenery) a następnie uruchamia się tylko i wyłącznie testowany komponent a jego otoczenie (inne procesy komunikujące się z nim w projekcie) symuluje. Cały test polega na bombardowaniu testowanego komponentu wiadomościami aby wymusić odpowiednie zachowanie. Dla przykładu istnieją 3 komponenty A, B oraz C, testowanym jest komponent B, wiadomo że w docelowym zachowaniu komponent A wysyła do komponentu B wiadomość x, a komponent B po wykonaniu kilku operacji wysyła wiadomość y do komponentu C. Programista testując komponent B powinien mu wstrzyknąć wiadomość x (której struktura w teście jest znana) za pomocą symulowanego komponentu A i sprawdzić czy w takim przypadku symulowany komponent C otrzyma wiadomość y. Jeśli tak się niestanie oznacza to, że jeszcze jakaś część komponentu B nie została zaimplementowana według wymagań, w przeciwnym wypadku test uznał pewną zależność komponentów za poprawną. Popularnymi językami w których istnieje możliwość pisania testów komponentowych są Python oraz ttcn3.

Istnieje również automatyzacja testów. Stosuje się ją w wypadku większych zmian aby mieć pewność iż wprowadzanie nowych funkcjonalności nie naruszyło poprzednich. Automatyzuje się przede wszystkim testy weryfikacyjne (SyVe[10]), które w standardowym (albo w większości procesów) polegają na manualnym testowaniu danych funkcjonalności i weryfikowaniu czy otrzymany został pożądany rezultat. Istotną rolę pełnią również testy integracyjne aplikowalne tylko do większych projektów, w których istotną role odgrywa współdziałanie wielu komponentów, z reguły implementowane przez różne zespoły.

Z punktu widzenia pracy najbardziej istotna jest weryfikacja oprogramowania z początkowymi założeniami (wymaganiami). Należy zadbać o powtarzalność testu i o przejrzystość formatu danych wyjściowych. Testy należy wykonać w środowisku docelowym (z zewnętrznym urządzeniem) aby uniknąć późniejszych wątpliwości. Należy również przygotować środowisko dla automatyzacji.

[9] tutaj w rozumieniu procesu
[10] SyVe - System Verification

III. Biblioteki udostępniające API do komunikacji z USB

Głównym celem projektu jest osiągniecie jak najszybszej prędkości przesyłu danych. Do realizacji celu konieczne jest użycie odpowiedniej biblioteki do komunikacji z interfejsem USB. W rozdziale przedstawiony został krótki opis najbardziej popularnych bibliotek posiadających interfejs komunikacyjny ze złączami USB.

III.1. Biblioteka libUSB

LibUSB jest biblioteką stworzoną w 2007 roku. Napisana w języku C pozwala na prosty i łatwy dostęp do urządzenia USB. Jest w 100% przeznaczona dla użytku developera. Biblioteka ma za zadanie ułatwić pisanie aplikacji opartych na komunikacji USB z mikrokontrolerem. Biblioteka libUSB jest przenośna a co za tym idzie dostępna na wiele platform (Linux, OS X, Windows, Android, OpenBSD, etc.) wraz z niezmiennym API. Nie są wymagane dodatkowe uprawnienia aby komunikacja z urządzeniem przebiegała poprawnie. Wspiera standardy USB:

- USB1.0
- USB1.1
- USB2.0
- USB3.0

Funkcjonalność biblioteki:

1. wszystkie typy transferu są wspierane (control, bulk, interrupt, isochronous)
2. dwa interfejsy
   (a) synchroniczny (prosty)
   (b) asynchroniczny (bardziej złożony)
3. stosowanie wątków jest bezpieczne
4. lekka biblioteka z prostym API
5. kompatybilna wstecznie (do wersji libUSB-0.1)

(informacje uzyskane dzięki [9])

III.2. Biblioteka winUSB

Microsoft Windows począwszy od systemu Windows Vista wprowadził nowy zestaw bibliotek umożliwiający developerom korzystnie z portów USB. WinUSB udostępnia proste API, które pozwala aplikacji na bezpośredni dostęp do portów USB. Został stworzony w gruncie rzeczy dla prostych urządzeń obsługiwanych tylko przez jedną aplikację takich jak urządzenia do odczytu wskaźników pogodowych czy też innych programów które potrzebują szybkiego i bezpośredniego dostępu do portu. WinUSB udostępnia API aby odblokować developera przy pracy z portami USB z poziomu user-mode. W Windowsie 7 USB Media Transfer Protocol (MTP) używa winUSB zamiast poprzednio stosowanych rozwiązań kernela (krenel mode filter driver).

Media Transfer Protocol jest rozszerzeniem PTP (Picture Transfer Protocol) i jest protokołem pozwalającym na przesyłanie atomowe plików audio oraz wideo z oraz do urządzenia. PTP początkowo został zaprojektowany do ściągania zdjęć, obrazów z aparatów cyfrowych, Media Transfer Protocol pozwala na przesyłanie plików muzycznych z cyfrowych urządzeń odtwarzających muzykę oraz pliki video z urządzeń pozwalających na ich odtworzenie.

MTP jest częścią frameworku "Windows Media" blisko związanym z odtwarzaczem Windows Media Player. Systemy Windows począwszy od Windows XP SP2 wspierają MTP. Windows XP wymaga Windows Media Player w wersji 10 lub wyższej, późniejsze wersje systemu wspierają już go domyślnie. Microsoft posiada dodatkowo możliwość zainstalowania MTP na wcześniejszych wersjach systemu ręcznie do wersji Microsoft Windows 98.

Twórcy standardu USB ustandaryzowali MTP jako pełnoprawną klasę dla urządzeń USB w maju 2008r. Od tamtej pory MTP jest oficjalnym rozszerzeniem PTP i współdzieli ten sam kod klasy. [10, 11, 12, 13]

Podsumowując w projekcie została użyta biblioteka libUSB. Głównym powodem była możliwość kompilacji programów zarówno pod systemami Unix jak i z rodziny Microsoft Windows bez jakichkolwiek (lub znikomych) zmian w kodzie.

IV. Opis zawartości biblioteki libUSB

Rozdział dokumentujący API używane w projekcie. W rozdziale III został uzasadniony wybór tej właśnie biblioteki. [8]

IV.1. najważniejsze struktury

IV.1.1. libusb_context

libusb_context jest strukturą reprezentującą sesję libusb.
Koncepcja indywidualnych sesji libusb pozwala aby program mógł korzystać z dwóch bibliotek (lub dynamicznie ładować dwa moduły) z których obie nie zależnie korzystają z libusb. To zapobiega ingerencji (interferencji) pomiędzy dwoma programami używającymi libusb. Dla przykładu libusb_set_debug() nie zaingeruje w działanie innego programu korzystającego z libusb, natomiast libusb_exit() nie wyczyści pamięci używanej przez inny program libusb.
Sesje tworzone są za pomocą libusb_init() oraz czyszczone za pomocą libusb_exit(). Jeśli zagwarantowane jest to, że dana aplikacja jest jedyną która korzysta z libusb, twórca jej nie musi przejmować się kontekstami (strukturą libusb_context), wystarczy aby przekazywał do wszystkich funkcji, gdzie struktura jest wymagana wartość NULL. Jest to równoważne z użyciem domyślnego kontekstu.

IV.1.2. libusb_device_handle

Struktura reprezentująca uchwyt do urządzenia USB.
Jest to nieprzejrzysty typ, użycie jest możliwe tylko za pomocą wskaźnika, zazwyczaj dostarczanego za pomocą funkcji libusb_open().
Uchwyt do urządzenia USB jest używany do wykonywania operacji wejścia/wyjścia. Po zakończeniu wszystkich operacji należy wywołać libusb_close().

IV.2. najważniejsze funkcje

IV.2.1. libusb_init

Inicjalizacja biblioteki.
Funkcja musi zostać wywołana przed wywołaniem jakiejkolwiek innej funkcji z biblioteki libUsb.

Jeśli w argumencie nie zostanie dostarczony żaden wskaźnik (wyjściowy) kontekstu, zostanie stworzony domyślny kontekst. W przypadku jeśli domyślny kontekst już istnieje zostanie on ponownie użyty (bez ponownej inicjalizacji).
Funkcja zwraca wartość 0 w wypadku powodzenia, w przeciwnym wypadku zwraca kod błędu.

IV.2.2. libusb_open_device_with_vid_pid

Wygodna funkcja służąca odszukaniu konkretnego urządzenia na podstawie jego vendorId oraz productId (są to parametry charakteryzujące każde urządzenie).
Ta funkcja jest używana w przypadkach kiedy z góry jest znane vendorId oraz productId. Najczęściej są to przypadki pisania aplikacji aby przetestować jakąś określoną funkcjonalność. Funkcja pozwala uniknąć wywołanie libusb_get_device_list() i dbania o odpowiednie czyszczenie pamięci po liście.
Pierwszym parametrem jest kontekst uzyskany za pomocą libusb_init().
Kolejnymi parametrami są vendorId oraz productId.
Funkcja zwraca uchwyt do znalezionego urządzenia, lub NULL w wypadku kiedy nie może znaleźć pożądanego urządzenia (o podanym productId oraz vendorId) lub błędu.

IV.2.3. libusb_kernel_driver_active

Funkcja sprawdzająca czy sterownik jądra kernela jest aktywny na interfejsie.
W wypadku kiedy sterownik jądra jest aktywny nie możliwe jest zgłoszenie użycia interfejsu, a co za tym idzie libUSB nie może wykonać operacji wejścia/wyjścia.
Funkcjonalność jest nie dostępna w systemie Windows.
Do funkcji należy przekazać uchwyt do urządzenia oraz numer interfejsu.
Funkcja zwraca wartość 0, jeśli żaden sterownik jądra nie jest aktywny, wartość 1 w wypadku jeśli istnieje aktywny sterownik jądra.
W wypadku błędów: LIBUSB_ERROR_NO_DEVICE jeśli urządzenie zostanie odłączone, LIBUSB_ERROR_NOT_SUPPORTED dla platform, gdzie funkcjonalność nie jest wspierana oraz inny kod błędu w wypadku innego błędu.

IV.2.4. libusb_detach_kernel_driver

Dzięki funkcji możliwe jest odłączenie sterownika jądra kernela od interfejsu.
Sukces operacji umożliwi zgłoszenie użycia interfejsu i wykonanie operacji wejścia/wyjścia.
Funkcjonalność nie jest dostępna dla systemu Windows.
Parametrami są uchwyt do urządzenia oraz numer interfejsu.

Funkcja zwraca wartość 0 w momencie powodzenia, LIBUSB_ERROR_NOT_FOUND jeśli żaden sterownik nie był aktywny, LIBUSB_ERROR_INVALID_PARAM jeśli interfejs nie istniej, LIBUSB_ERROR_NO_DEVICE jeśli urządzenie zostało odłączone, LIBUSB_ERROR_NOT_SUPPORTED dla platform, gdzie funkcjonalność nie jest wspierana lub inny kod błędu w wypadku inne błędu.

IV.2.5. libusb_claim_interface

Dzięki funkcji libusb_claim_interface() możliwe jest zgłoszenie użycia danego interfejsu dla danego urządzenia.
Wywołanie funkcji jest wymagane przed wykonaniem operacji wejścia/wyjścia dla dowolnego punktu końcowego interfejsu.
Jest dozwolone wywołanie funkcji dla interfejsu już wcześniej zgłoszonego, w tym wypadku zostanie zwrócona wartość 0 bez wykonywania żadnych operacji.
W wypadku jeśli zmienna auto_detach_kernel_driver jest ustawiona na wartość 1 dla danego urządzenia (zmienna ustawiana za pomocą funkcji: libusb_set_auto_detach_kernel_driver()) sterownik jądra zostanie odłączony (jeśli to konieczne), w przypadku niepowodzenia zostanie zwrócony błąd odłączenia.
Sama procedura wewnątrz funkcji nie jest skomplikowana, nie wymaga wysyłania czegokolwiek po magistrali. Są to proste instrukcje mówiące systemowi operacyjnemu iż aplikacja chce korzystać z danego interfejsu.
Nie jest to funkcja blokująca.
Parametrami są uchwyt do urządzenia oraz numer interfejsu zgłaszanego.
Wartość 0 zostaje zwrócona w wypadku powodzenia operacji.
W wypadku niepowodzenia kody błędu t.j. LIBUSB_ERROR_NOT_FOUND jeśli podany interfejs nie istnieje, LIBUSB_ERROR_BUSY jeśli inny program lub sterownik zarezerwował dany interfejs, LIBUSB_ERROR_NO_DEVICE jeśli urządzenie zostało odłączone oraz inne.

IV.2.6. libusb_bulk_transfer

Dzięki funkcji możliwy jest przesył większej grupy danych.
Kierunek jest określony na podstawie bitów kierunkowych punktu końcowego interfejsu.
Dla odczytu, jeden z parametrów określa ilość danych jaka jest spodziewana przy odczycie. Jeżeli odebrana zostanie mniejsza ilość danych niż oczekiwana, funkcja po prostu zwróci te dane wraz z dodatkowym parametrem określającym ilość otrzymanych danych. Istotne przy odczycie jest to aby sprawdzić czy ilość oczekiwanych danych jest taka jak odczytana.
W wypadku zapisu również należy sprawdzić czy ilość danych wysłanych pokrywa się z ilością danych skierowaną do wysyłki.

Wskazana jest również weryfikacja ilości danych wysłanych/odebranych w wypadku wystąpienia timeoutu (funkcja zwróci kod błędu określający timeout). libUSB może podzielić wysyłane dane na mniejsze części i timeout może wystąpić po wysłaniu kilku z nich. Ważne jest to, że nie oznacza to iż nic nie zostało wysłane/odebrane, dlatego należy sprawdzić ilość elementów wysłanych/odebranych i dostosować odpowiednio kolejne kroki.
Funkcja przyjmuje następujące parametry: uchwyt urządzenia z którym aplikacja będzie się komunikować, adres punktu końcowego interfejsu po którym będzie odbywała się komunikacja, wskaźnik do pamięci danych która ma zostać przetransferowana (w wypadku zapisu) lub odebrana (w wypadku odczytu), ilość danych do wysłania (w przypadku zapisu) lub oczekiwana ilość danych do odebrania (w przypadku odczytu), ilość danych przetransferowanych (w obu przypadkach), maksymalna długość czasu na wykonanie operacji, dla nieograniczonego należy użyć wartości równej 0.
W przypadku poprawności działania funkcja zwraca wartość 0 oraz ilość przetransferowanych danych przekazanych do funkcji za pomocą wskaźnika.
W przeciwnym wypadku funkcja zwraca: LIBUSB_ERROR_TIMEOUT jeśli transfer przekroczył określony czas, LIBUSB_ERROR_PIPE jeśli wystąpił błąd związany z punktem końcowym, LIBUSB_ERROR_OVERFLOW jeśli urządzenie wysłało więcej danych niż przewidziane w buforze, LIBUSB_ERROR_NO_DEVICE jeśli urządzenie zostało odłączone oraz inne.

IV.2.7. libusb_release_interface

Funkcja zwalnia rezerwacje wcześniej zgłoszonego interfejsu za pomocą funkcji libusb_claim_interfa
Zwolnienie wszystkich interfejsów jest wymagane przed zamknięciem urządzenia.
Nie jest to blokująca funkcja.
W wypadku jeśli zmienna auto_detach_kernel_driver jest ustawiona na wartość 1 dla danego urządzenia (zmienna ustawiana za pomocą funkcji: libusb_set_auto_detach_kernel_driver()) sterownik jądra zostanie ponownie podłączony zaraz po zwolnieniu interfejsu.
Parametrami są: uchwyt urządzenia oraz numer interfejsu dla niego poprzednio zarezerwowanego.
Metoda zwraca 0 gdy wszystkie operacje się powiodą.
W przeciwnym wypadku zwraca: LIBUSB_ERROR_NOT_FOUND jeśli interfejs nie został poprzednio zarezerwowany (zgłoszony do użycia), LIBUSB_ERROR_NO_DEVICE jeśli urządzenie zostało odłączone oraz inne.

IV.2.8. libusb_close

Zwalnia uchwyt do urządzenia.
Funkcja powinna być wołana na wszystkich używanych uprzednio uchwytach.

Funkcja pokrótce niszczy referencje stworzoną za pomocą libus_open() dla danego urządzenia.
Jest to funkcja nie blokująca.
Parametrem jest uchwyt przeznaczony do zamknięcia.

IV.2.9. libusb_exit

Funkcja zamyka dostęp do biblioteki.
Powinna być wołana po zamknięciu wszystkich otwartych urządzeń ale przed zakończeniem działania programu.
Parametrem jest contekst który ma zostać zamknięty, w wypadku wartości NULL wybierany jest domyślny.

IV.2.10. libusb_alloc_transfer

Funkcja przygotowuje transfer z wyspecyfikowaną ilością izochronicznych deskryptorów pakietów.
Funkcja zwraca uchwyt do zainicjalizowanego transferu. Kiedy wszystkie operacje zostaną na nim wykonane należy wywołać libusb_free_transfer().
Transfer przygotowywany dla nie izochronicznego punktu końcowego należy wywoływać z wartością zero jako ilość izochronicznych pakietów.
Dla transferów izochronicznych wymagane jest podanie prawidłowej wartości deskryptorów pakietów jakie mają zostać zalokowane w pamięci. Zwracany transfer nie jest domyślnie zainicjalizowany jako izochroniczny, wymagane dodatkowo jest ustawienie pola libusb_iso_packets oraz type.
Alokowanie transferu jako izochroniczny (z podana ilością deskryptorów pakietów do zalokowania) a następnie używanie transferu jako nie izochroniczny jest w 100% bezpieczne ale pod warunkiem jeśli pole num_iso_packets jest ustawione na zero oraz pole type jest ustawione prawidłowo.
Funkcja jako prametr przyjmuje ilość izochronicznych deskryptorów pakietów.
Zwracaną jest zalokowany transfer lub NULL w wypadku błędu.

IV.2.11. libusb_fill_bulk_transfer

Funkcja pozwala na łatwe przygotowanie struktury libusb_transfer na transfer masowy.
Parametrami są:

- uchwyt do ustawianego transferu
- uchwyt do urządzenia dla którego ustawiany jest transfer
- adres punktu końcowego gdzie dane mają zostać wysłane

- bufor danych do wysłania/odebrania
- długość (wielkość) wysyłanych/odbieranych danych
- wskaźnik do funkcji która ma się wywołać po zakończeniu transferu (callback)
- dodatkowe dane, które programista może opcjonalnie wysłać do funkcji wywołanej po zakończeniu transferu
- czas oczekiwania w milisekundach na zakończenie transferu

IV.2.12. libusb_submit_transfer

Funkcja wykonuje podany (ustawiony) transfer.
Funkcja wykonuje operacje na interfejsie po czym bezzwłocznie kończy działanie.
Jedynym parametrem jaki przyjmuje funkcja jest wskaźnik do transferu jaki ma zostać wykonany.
W wypadku kiedy wszystko się powiedzie funkcja zwraca wartość równą 0, w pozostałych przypadkach zwraca kod błędu tj. LIBUSB_ERROR_NO_DEVICE w wypadku kiedy urządzenie nie jest podłączone, LIBUSB_ERROR_BUSY w przypadku jeśli akcja została już wykonana, LIBUSB_ERROR_NOT_SUPPORTED jeśli flagi transferu (ustawienia) są nie wspierane przez system operacyjny oraz inne.

IV.2.13. libusb_free_transfer

Funkcja odpowiada za zwolnienie pamięci zajętą przez strukturę libusb_transfer
Funkcja ta powinna być zawołana dla wszystkich transferów zaalokowanych przez libus_alloc_transfer().
Jeśli dodatkowo flaga LIBUSB_TRANSFER_FREE_BUFFER jest ustawiona oraz bufor transferu jest nie zerowy, pamięć po nim zostanie również zwolniona za pomocą standardowej funkcji alokującej (np. free()).
Dozwolone jest zawołanie funkcji z parametrem równym NULL, w takim przypadku funkcja zakończy się bez błędu (ale nic nie zostanie zwolnione).
Nie dozwolone jest zwalnianie pamięci po nie zakończonym (aktywnym) jeszcze transferze, czyli takim który wystartował ale jeszcze nie wywołany został callback do niego (lub timeout).
Parametrem funkcji jest wskaźnik do transferu do zwolnienia.

V. Urządzenie testowe

Aby zrealizować projekt konieczne było użycie dodatkowego hardwaru, którego zadaniem było odbieranie danych po przeciwnej stronie jak i przesył w drugą stronę. Inwestygacja rozpoczęła się od bardzo prostych urządzeń takich jak raspberry pi B+, a zakończyła się na mikrokontrolerze LandTiger.

Rys. V.1. *Mikrokontroler LandTiger wraz z wyświetlaczem*

Mikrokontroler LandTiger oparty na LPC1768 został wyprodukowany przez firmę PowerMCU i można ją zakupić od wielu dostawców na eBay lub innych serwisach świadczących usługi zakupów przez internet. Średni koszt waha się w granicach $70 za płytkę wraz z wyświetlaczem LCD 3,2 cala o rozdzielczości 320x240 pikseli, z zasilaczem oraz zestawem kabli. [14]
Funkcjonalności:

(a) 2 porty RS232, jeden z nich wspiera ISP (In-system Programming)

(b) 2 interfejsy magistrali CAN (Controller Area Network)

(c) interfejs RS485

(d) interfejs Ethernetowy RJ45-10/100M

(e) przetwornik cyfrowo-analogowy (DAC) wraz wmontowanym głośnikiem (wyjściem interfejsu) oraz sterownikiem dźwięku (LM386)

(f) przetwornik analogowo-cyfrowy (ADC) wraz z wbudowanym potencjomentrem (wejściem interfejsu).

(g) Kolorowy 3,2 cala (lub 2,8 cala) dotykowy wyświetlacz LCD o rozdzielczości 320x240 pikseli.

(h) interfejs USB2.0 (USB Host oraz USB Device)

(i) interfejs kard SD/MMC

(j) interfejs I2C połączony z 2Kbit pamięcią EEPROM

(k) interfejs SPI połączony z 16Mbit pamięcią flash

(l) 2 user keys, 2 function keys

(m) 8 diód typu LED

(n) pięciokierunkowy joystick

(o) wsparcie dla pobierania ISP

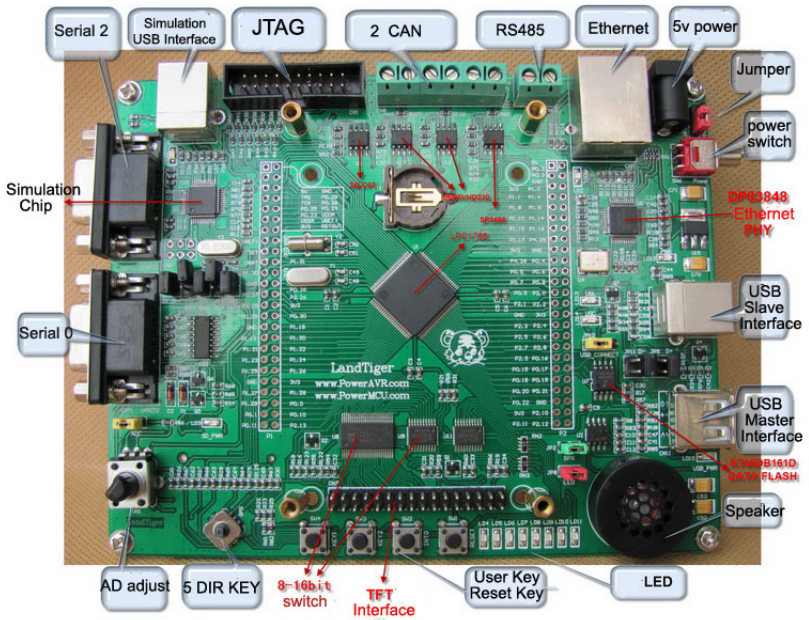
(p) pobieranie z użyciem JTAG, interfejs dla debugowania

(q) zintegrowany emulator kompilacji JLINK - wspiera możliwość debugownia online (po kablu USB podłączonym do PC) dla środowisk deweloperskich tj. KEIL, IAR, CooCox i innych

(r) dodatkowe 5V port zasilający (możliwe jest też za pomocą portu USB

LandTiger jest oparty na LPC1768. Wbudowany hardware wspiera ISP aby umożliwić załadowanie kodu (z użyciem bin2hex oraz flashmagic).
Alternatywą jest to, że kod może zostać załadowany za pomocą emulatora JLINK JTAG/SWD lub za pomocą zewnętrznego urządzenia JTAG.
Port COM1 (UART0) wspiera komunikacje z PC w obie strony. Wszelkie funkcje portu USB są wspierane z minimalnymi zmianami w oprogramowaniu. Podobnie jest z Ethernetem, z niewielkimi zmianami w oficjalnym kodzie dla LPC1768 kod jest w stanie się uruchomić na LandTigerze.

Wyświetlacz LCD jest oparty na kontrolerze SSD1289. Wyświetlacz może zostać odłączony od płyty. Używa 8-bitowej magistrali $P2_0..P2_7$. Kontroler ekranu dotykowego jest dostarczony razem z modułem wyświetlacza. Interfejs pomiędzy ekranem dotykowym a LPC1768



Rys. V.2. *Mikrokontroler LandTiger wraz z opisem poszczególnych elemetnów*

jest możliwy dzięki SPI.
Główne różnice pomiędzy LandTigerem a LPC1768:

(a) płyta po podłączeniu do PC nie pokazuje się jako zewnętrzne urządzenie magazynujące

(b) aby ściągnąć nowe pliki binarne należy użyć ISP lub JTAG

(c) brak wsparcia dla serialowego portu po linku USB, należy używać RS232 lub portu USB

(d) brak wsparcia dla logicznego systemu plików

(e) brak wsparcia dla 4 diód typu LED (istnieje możliwość użycia inncyh).

Parametry mikrokontrolera LandTiger wydawały się być zadowalające aby wykonać na nim testy. Niestety jak się podczas egzekucji testów okazało (a zostało dokładniej opisane w rozdziale VII) pomimo wlutowanego złącza USB wspierającego USB2.0 chip na kontrolerze wspiera standard USB1.1. Dzięki wynikom uzyskanym z użyciem LandTiger'a możemy obliczyć możliwości gdyby wsparcie USB2.0 na kontrolerze było zaimplementowane.

VI. Implementacja

Implementacja opiera się na wykorzystaniu prostych podstawowych funkcji z biblioteki libUsb. Rozwiązania zostały obudowane w odpowiednie klasy wraz z zastosowaniem polimorfizmu aby uzyskać możliwość łatwego dodawania nowych udoskonaleń.

VI.1. Klasa Mode

Klasa Mode jest nadzbiorem funkcjonalności. Klasa ta obudowuje podstawowe funkcjonalności takie jak:

- inicjalizacja
  - pobieranie kontekstu
  - pobieranie uchwytu urządzenia
  - rejestracja interfejsu
- oczyszczanie pamięci

Klasa zawiera również czysto wirtualną metodę doTest(), której odpowiednie implementacje w klasach dziedziczących dostarcza funkcjonalności testom.

```
class Mode
{
public:
  Mode(int bufforSize, unsigned count, int vid, int pid, bool printOnlyResult) :
    _bufforSize(bufforSize), _count(count), _vid(vid), _pid(pid), _printOnlyResult(
    printOnlyResult)
  {
    _debugPrinter.set(!printOnlyResult);
  }
  virtual ~Mode();
  virtual int generateSymulatedData(unsigned char*, const int);
  virtual void printFinalInformation();
  virtual void initProcedures();
  virtual void getContext();
  virtual void getDeviceHandle();
  virtual void proceedWithInitUsb();
  virtual void closeLibUsb();
  virtual int doTest() = 0;

protected:
  int _bufforSize;
  unsigned _count;
  int _vid;
  int _pid;
```

```
  double _timeResult;
  DebugPrinter _debugPrinter;
  bool _printOnlyResult;
  libusb_context* _ctx;
  libusb_device_handle* _dev_handle;

};
```

Listing VI.1. *Deklaracja klasy Mode*

VI.1.1. Metoda getContext

W listingu VI.2 przedstawione zostało ciało metody getContext() klasy Mode.
Metoda nie przyjmuje żadnego argumentu, natomiast inicjalizuje kontekst i zapamiętuje go jako składnik w składowej klasy. W rozdziale IV wspomniane zostało iż możliwe jest używanie kontekstu domyślnego, wtedy zamiast użycia libusb_init() należałoby przypisać wskaźnikowi do kontekstu wartość równą NULL.
Aby używać domyślnego kontekstu należy mieć pewność iż aplikacja/wątek jest jedynym użytkownikiem libUSB.
W wypadku niepoprawnego działania wyrzucany jest błąd (ważne aby był przechwycony w funkcji main).

```
void Mode::getContext()
{
  int r = libusb_init(&_ctx);
  if(r < 0) {
    throw std::runtime_error("Init Context error");

  }
}
```

Listing VI.2. *Metoda Mode::getContext()*

VI.1.2. Metoda getDeviceHandler

Metoda nie przyjmuje żadnego argumentu. Ustawiany jest uchwyt do urządzenia na podstawie jego vendorId oraz productId (testy były wykonane na mikrokontrolerze LandTige). W wypadku niepowodzenia zostanie zgłoszony odpowiedni błąd (np. braku podłączonego i uruchomionego mikrokontrolera). Ciało funkcji zostało przedstawione w Listingu VI.3

```
void Mode::getDeviceHandle()
{
  _dev_handle = libusb_open_device_with_vid_pid(_ctx, _vid, _pid);
```

```
  if(_dev_handle == NULL)
    throw std::runtime_error("Cannot open device!");
  else
    _debugPrinter << "Device Opened\n";
}
```

Listing VI.3. *Metoda Mode::getDeviceHandler()*

VI.1.3. Metoda proceedWithInitLibUsb

Metoda odpowiedzialna za dokończenie procedur inicjalizacyjnych. Ciało metody zostało przedstawione w Listingu VI.4. Metoda sprawdza dodatkowo czy sterownik jądra kernela jest aktywny, dla platformy Windows funkcja zwróci wartość LIBUSB_ERROR_NOT_SUPPORTED (!= 1) i zachowa się analogicznie jak w wypadku nie aktywnego sterownika w systemie UNIX (warunek bedzie nie spełniony).
Kolejnym krokiem jest rezerwacja przez program konkretnego interfejsu za pomocą funkcji libusb_claim_interface.
W wypadku niepoprawnego działania metoda wyrzuci runtime_error który należy obsłużyć w funkcji main().

```
void Mode::proceedWithInitUsb()
{
  if(libusb_kernel_driver_active(_dev_handle, 0) == 1) { //find out if kernel driver
    is attached
    _debugPrinter << "Kernel Driver Active\n";
    if(libusb_detach_kernel_driver(_dev_handle, 0) == 0) //detach it
      _debugPrinter << "Kernel Driver Detached!\n";
  }
  int status = libusb_claim_interface(_dev_handle, 1);
  if(status < 0)
  {
    throw std::runtime_error("Cannot Claim Interface");
  }
  _debugPrinter << "Claimed Interface\n";
}
```

Listing VI.4. *Metoda Mode::proceedWithInitLibUsb()*

VI.1.4. Metoda initProcedures()

Metoda przedstawiona w listingu VI.5 odpowiada za wywołanie procedur inicjalizacji w odpowiedniej kolejności.

```
void Mode::initProcedures()
{
  getContext();
  getDeviceHandle();
  proceedWithInitUsb();
}
```

Listing VI.5. *Metoda Mode::initProcedures()*

VI.1.5. Metoda closeLibUsb

Metoda przedstawiona w Listingu VI.6 odpowiada za zwolnienie interfesju oraz zasobów uprzednio zajętych na czas testu.

```
void Mode::closeLibUsb()
{
  int status = libusb_release_interface(_dev_handle, 1);
  if(status != 0) {
    throw std::runtime_error("Cannot Relase Interface");
  }
  _debugPrinter << "Released Interface\n";
  libusb_close(_dev_handle);
  libusb_exit(_ctx);
}
```

Listing VI.6. *Metoda Mode::closeLibUsb()*

VI.1.6. Metoda printFinalInformation

Metoda mająca na zadanie wypisanie ostatecznych wyników. Wyniki mogą zostać przedstawione jako czytelna zwykła informacja dla użytkownika lub w postaci kolumnowej (używanej przy tworzeniu wykresów).

```
void Mode::printFinalInformation()
{
  _debugPrinter << "Sending of: " << _bufforSize * _count << "Bytes using bufferSize="
     << _bufforSize << " takes " << _timeResult << "s.\n";
  if(_printOnlyResult)
  {
    unsigned allSendReceivedData = _bufforSize * _count;
    double sendingTime = _timeResult/2;
    double receivingTime = _timeResult/2;
```

```
    printf("%d\t%u\t%u\t%.2f\t%.2f\t%.2f\t%.2f\t%.2f\t%.2f\n", _bufforSize, _count,
    allSendReceivedData, _timeResult, sendingTime, receivingTime,
              allSendReceivedData/_timeResult, allSendReceivedData/sendingTime,
    allSendReceivedData/receivingTime);
  }
}
```

Listing VI.7. *Metoda Mode::printFinalInformation()*

VI.2. Klasa SynchMode

Klasa SynchMode odpowiada odpowiednią konfigurację USB dla przesyłu synchronicznego.

```
class SynchMode : public Mode
{
public:
  SynchMode(int bufforSize, unsigned count, int vid, int pid, bool printOnlyResult);
  virtual ~SynchMode();
  virtual int doTest() override;

private:

};
```

Listing VI.8. *Deklaracja klasy SynchMode*

Klasa używa wszystkich domyślnych ustawień (nie przeciąża żadnej metody wirtualnej), oczywiście metoda doTest() wymaga implementacji.

VI.2.1. Metoda doTest

Metoda w całości przedstawiona w listingu VI.9 i jest odpowiedzialna za wykonanie podstawowego pomiaru czasu przepływu danych w obie strony pomiędzy PC a mikrokontrolerem.
Została zaprojektowana w taki sposób aby konkretny bufor danych został wysłany oraz odebrany określoną ilość razy i zwrócony przedział czasowy w jakim udało się to uzyskać. Powodem wysyłania/odbierania danych określoną ilość razy jest fakt dość sporych ograniczeń jeśli chodzi o chip wbudowany w płytę LandTiger.

```
int SynchMode::doTest()
{
  unsigned char *data_out = new unsigned char[_bufforSize]; //data to write
  unsigned char* data_in = new unsigned char[_bufforSize];
```

```
  generateSymulatedData(data_out, _bufforSize);
  int howManyBytesIsSend;
  int howManyBytesReceived;

  time_t start_t, end_t;
      _timeResult = 0;


  time(&start_t);
  for(int i = 0; i < _count; ++i)
  {
    int sendStatus = libusb_bulk_transfer(_dev_handle, (2 | LIBUSB_ENDPOINT_OUT),
    data_out, _bufforSize, &howManyBytesIsSend, 0);
    if(sendStatus == 0 && howManyBytesIsSend == _bufforSize)
    {
      //here was printing data for debugging only
    }
    else
    {
      delete [] data_out;
      throw std::runtime_error("Write Error!");
    }


    int readStatus = libusb_bulk_transfer(_dev_handle, (2 | LIBUSB_ENDPOINT_IN),
    data_in, _bufforSize * sizeof(unsigned char), &howManyBytesReceived, 0);
    if (readStatus == 0 && howManyBytesReceived == howManyBytesIsSend)
    {
      //here was printing data for debugging only
    }
    else
    {
      delete[] data_in;
      throw std::runtime_error("Read Error! ");
    }

  }
  time(&end_t);
  _timeResult = difftime(end_t, start_t);
  delete[] data_in;
  delete [] data_out;
  return 0;
}
```

Listing VI.9. *Metoda SynchMode::doTest()*

VI.3. Klasa AsynchMode

Klasa AsynchMode odpowiada odpowiednią konfigurację USB dla przesyłu asynchronicznego.

```
class AsynchMode : public Mode
{
public:
  AsynchMode(int bufforSize, unsigned count, int vid, int pid, bool printOnlyResult);
  virtual ~AsynchMode();
  virtual int doTest() override;
  virtual void initProcedures() override;
  void closeLibUsb() override;
private:
  pthread_mutex_t _sender_lock;
  pthread_mutex_t _receiver_lock;
  pthread_cond_t _sender_cond;
  pthread_cond_t _receiver_cond;

  libusb_transfer* _senderTransfer;
  libusb_transfer* _receiverTransfer;
  pthread_t _receiverThread;

};
```

Listing VI.10. *Deklaracja klasy SynchMode*

Klasa zawiera wzbogaconą inicjalizację o alokacje transferów (wysyłającego oraz odbierającego) oraz inicjalizację mutexów.
Dosyć istotnym elementem jest przedstawiony w listingu VI.11 dodatkowy namespace ułatwiający obsługę callbacków. Znajduje się w nim zarówno obsługa wątku nasłuchującego jak i obsługa poszczególnych zdarzeń.

```
namespace ThreadHelper
{
  struct TransferStatus
  {
    time_t startTest;
    time_t stopTest;
    int allCompleted;
    libusb_transfer* senderHandler;
    libusb_transfer* receiverHandler;
    int sendCount;
    int receivedCount;
    int needToBeSendReceived;
    libusb_context* ctx;
```

```
  int waitForSender;
  int waitForReceiver;
  int particularSendComplete;
  int particularReceiveComplete;
  pthread_mutex_t* sender_lock;
  pthread_mutex_t* receiver_lock;
  pthread_cond_t* sender_cond;
  pthread_cond_t* receiver_cond;
};

static void LIBUSB_CALL cb_read(struct libusb_transfer *transfer)
{
  if (transfer->status != LIBUSB_TRANSFER_COMPLETED) {
    std::cerr << "Reading data error!" << std::endl;
    return ;
  }
  TransferStatus* ts = static_cast<TransferStatus*>(transfer->user_data);
  ts->receivedCount++;
  ts->particularReceiveComplete = 1;

  pthread_mutex_lock(ts->sender_lock);
  ts->waitForSender = 0;
  pthread_cond_signal(ts->sender_cond);
  pthread_mutex_unlock(ts->sender_lock);
  if(ts->receivedCount == ts->needToBeSendReceived)
  {
    time(&ts->stopTest);
    ts->allCompleted = 1;
  }

}

static void LIBUSB_CALL cb_send(struct libusb_transfer *transfer)
{
  if (transfer->status != LIBUSB_TRANSFER_COMPLETED) {
    std::cerr << "Sending data error!" << std::endl;
    return ;
  }
  TransferStatus* ts = static_cast<TransferStatus*>(transfer->user_data);
  ts->sendCount++;
  pthread_mutex_lock(ts->receiver_lock);
  ts->waitForReceiver = 0;
  ts->particularSendComplete = 1;
  pthread_cond_signal(ts->receiver_cond);
  pthread_mutex_unlock(ts->receiver_lock);
```

```
  }

  void* receiverThread(void* arg)
  {
    TransferStatus* transferStatus = static_cast<TransferStatus*>(arg);
    while(transferStatus->allCompleted != 1)
    {
      pthread_mutex_lock(transferStatus->receiver_lock);
      while(transferStatus->waitForReceiver) {
        pthread_cond_wait(transferStatus->receiver_cond, transferStatus->receiver_lock
    );
      }
      transferStatus->waitForReceiver = 1;

      pthread_mutex_unlock(transferStatus->receiver_lock);
      transferStatus->particularReceiveComplete = 0;
      libusb_submit_transfer(transferStatus->receiverHandler);
      while(transferStatus->particularReceiveComplete == 0)
        libusb_handle_events(transferStatus->ctx);
    }
  }

}
```

Listing VI.11. *Deklaracja klasy SynchMode*

VI.3.1. Metoda initProcedures oraz closeLibUsb

W wypadku Asynchronicznego przesyłania danych należy wykonać kilka dodatkowych operacji inicjalizacyjnych. Są to alokacje transferów oraz inicjalizacja mutexów. Całość widoczna w listingu **??**.

```
void AsynchMode::initProcedures()
{
  Mode::initProcedures();
  _senderTransfer = libusb_alloc_transfer(0);
  _receiverTransfer = libusb_alloc_transfer(0);
  if(_senderTransfer == NULL || _receiverTransfer == NULL)
  {
    throw std::runtime_error("Transfer allocation Error");
  }

  if(pthread_mutex_init(&_sender_lock, NULL) != 0 || pthread_mutex_init(&
    _receiver_lock, NULL) != 0)
```

```
  {
    throw std::runtime_error("Mutex Init Failed!");
  }

}
```

Listing VI.12. *Metoda AsynchMode::initProcedures*

Analogicznie sytuacja przedstawia się sytuacja w wypadku zwalniania zasobów. Dodatkową czynnością jest usunięcie mutexów oraz zwolnienie transferów. Całość dostępna w listingu VI.13.

```
void AsynchMode::closeLibUsb()
{
  pthread_mutex_destroy(&_sender_lock);
  pthread_mutex_destroy(&_receiver_lock);
  libusb_free_transfer(_senderTransfer);
  libusb_free_transfer(_receiverTransfer);
  Mode::closeLibUsb();
}
```

Listing VI.13. *Metoda AsynchMode::closeLibUsb*

VI.3.2. Metoda doTest

Podobnie jak w wypadku Synchronicznego mechanizmu tak i teraz należy dostarczyć implementacje metody doTest(). Jej działanie jest znacznie odmienne, polega na wystartowaniu drugiego wątku nasłuchującego odpowiedzi. Całość synchronizowana jest za pomocą mutexów (listing VI.14 oraz VI.11).

```
int AsynchMode::doTest()
{
  unsigned char *data_out = new unsigned char[_bufforSize]; //data to write
  unsigned char *data_in = new unsigned char[_bufforSize]; //data to read
  generateSymulatedData(data_out, _bufforSize);
  int howManyBytesIsSend;
  int howManyBytesReceived;
  ThreadHelper::TransferStatus transferStatus;
  transferStatus.allCompleted = 0;

  transferStatus.sendCount = 0;
  transferStatus.receivedCount = 0;
  transferStatus.senderHandler = _senderTransfer;
  transferStatus.receiverHandler = _receiverTransfer;
  transferStatus.needToBeSendReceived = _count;
```

```
  transferStatus.ctx = _ctx;
  transferStatus.waitForSender = 0;
  transferStatus.waitForReceiver = 1;
  transferStatus.sender_lock = &_sender_lock;
  transferStatus.sender_cond = &_sender_cond;
  transferStatus.receiver_lock = &_receiver_lock;
  transferStatus.receiver_cond = &_receiver_cond;

  libusb_fill_bulk_transfer(_senderTransfer, _dev_handle, (2 | LIBUSB_ENDPOINT_OUT),
    data_out, _bufforSize, ThreadHelper::cb_send, &transferStatus, 0);
  libusb_fill_bulk_transfer(_receiverTransfer, _dev_handle, (2 | LIBUSB_ENDPOINT_IN),
    data_in, _bufforSize, ThreadHelper::cb_read, &transferStatus, 0);

  _timeResult = 0.0;
  if(pthread_create(&_receiverThread, NULL, ThreadHelper::receiverThread, &
    transferStatus) != 0)
  {
    throw std::runtime_error("Error creating listener thread.");
  }

  time(&transferStatus.startTest);
  for(int i = 0; i < _count; ++i)
  {
    pthread_mutex_lock(&_sender_lock);
    while(transferStatus.waitForSender){
      pthread_cond_wait(&_sender_cond, &_sender_lock);
    }
    transferStatus.waitForSender = 1;

    pthread_mutex_unlock(&_sender_lock);
    transferStatus.particularSendComplete = 0;
    libusb_submit_transfer(_senderTransfer);
    while(transferStatus.particularSendComplete == 0)
      libusb_handle_events(_ctx);

  }
  pthread_join(_receiverThread, NULL);
  _timeResult = difftime(transferStatus.stopTest, transferStatus.startTest);
  delete data_out;
  delete data_in;
  return 0;
}
```

Listing VI.14. *Metoda AsynchMode::doTest()*

VI.4. Korzystanie z poszczególnych interfejsów

Kod przedstawiony w Listingu VI.15 doskonale ukazuje prostotę korzystania z libUSB. Użytkownik zobligowany jest do wprowadzenia wielkości bufora danych oraz ilości powtórzeń określających ile razy dany bufor zostanie wysłany do mikrokontrolera. Następnie zostaje wykonana inicjalizacja z użyciem wyżej wymienionych interfejsów (synchronicznego lub asynchronicznego na podstawie parametru).

```
int main(int argc, char* argv[])
{

  if(argc < 4)
  {
    std::cout << "use: " << argv[0] << " [S/A] <bufforSize> <fullDataSize> optional:<
    printOnlyResult>" << std::endl;
    std::cout << "First parameter tells if application should be run in Synchronous
    mode or Asynchronous" << std::endl;
    std::cout << "Note that max buffer of LandTiger is " << BUFFOR_MAX << "Bytes" <<
    std::endl;
    return 0;
  }
  bool printOnlyResult = false;
  if(argc == 5)
    printOnlyResult = argv[4][0] == '0' ? false : true;

  DebugPrinter debugPrinter(! printOnlyResult);
  char selectedMode = std::toupper(argv[1][0]);
  if(selectedMode != 'A' && selectedMode != 'S')
  {
    debugPrinter << "mode set is different than A or S, setting synchronous as default
    .\n";
    selectedMode = 'S';
  }

  unsigned bufforSize = atoi(argv[2]);
  if(bufforSize > BUFFOR_MAX)
  {
    debugPrinter << "bufforSize is grather than 64B, setting 64 as default\n";
    bufforSize = BUFFOR_MAX;
  }
  unsigned fullDataSize = atoi(argv[3]);
  unsigned count = fullDataSize/bufforSize;
  if(fullDataSize % bufforSize != 0)
  {
    debugPrinter << "It is impossible to send " << fullDataSize << "B using " <<
```

```
    bufforSize << "B, to simplify application work sending " << (++count) * bufforSize
     << "B\n";
  }


  debugPrinter << "Total size to send/receive: " << bufforSize << " x " << count << "
    = " << bufforSize * count << " Bytes\n";
  try
  {
    std::unique_ptr<Mode> mode;

    if(selectedMode == 'A')
      mode.reset(new AsynchMode(bufforSize, count, LAND_TIGER_VID, LAND_TIGER_PID,
    printOnlyResult));
    else
      mode.reset(new SynchMode(bufforSize, count, LAND_TIGER_VID, LAND_TIGER_PID,
    printOnlyResult));
    mode->initProcedures();
    mode->doTest();
    mode->printFinalInformation();
    mode->closeLibUsb();
  } catch (std::runtime_error& e)
  {
    std::cout << e.what() << std::endl;
  }

  return 0;
}
```

Listing VI.15. *Przykład użycia interfejsu synchronicznego lub asynchronicznego w zależności od parametryzacji*

VI.4.1. Parametryzacja

```
use: libusb [S/A] <bufforSize> <fullDataSize> optional:<printOnlyResult>
```

Listing VI.16. *uruchomienie testu*

- S/A - należy wybrać czy chcemy wykonać test w trybie Synchronicznym czy Asynchronicznym
- bufforSize - rozmiar bufforu danych jaki chcemy przeznaczyć dla jednej paczki danych [B]
- fullDataSize - całkowity rozmiar danych do wysłania [B]

- printOnlyResult - opcjonalny parametr w wypadku ustawienia na wartość '1' pozwala na łatwe zebranie wyników bez zbędnych informacyjnych wiadomości (same dane).

VI.5. Embedded code

Po stronie mikrokontrolera LandTiger konieczne było zaimplementowanie prostego stuba mającego za zadanie krótkie potwierdzenie odebrania wiadomości (jako reakcja na zdarzenie) lub wykonanie akcji na kontrolerze. Poniższy listing zawiera przykładowy kod wysyłający jako potwierdzenie otrzymane dane (niestety aby nie wystąpił w wypadku większej ilości przysłanych danych, wysyłana jest zawartość ograniczona do 64 B). [1, 2, 4]

```
void USB_Receiver_Sender()
{
  static char serBuf [USB_CDC_BUFSIZE];
  int  numBytesToRead, numBytesRead, numAvailByte;

  USB_Init();
  USB_Connect(TRUE);                          // USB Connect
  while (!USB_Configuration) ;                // wait until USB is configured
  while(1)
  {
    CDC_OutBufAvailChar (&numAvailByte);
    if (numAvailByte > 0)
    {
        numBytesToRead = numAvailByte > MAX_TEMP_BUFFOR ? MAX_TEMP_BUFFOR :
    numAvailByte;
        numBytesRead = CDC_RdOutBuf (&serBuf[0], &numBytesToRead);
        USB_WriteEP (CDC_DEP_IN, (unsigned char *)&serBuf[0], numBytesRead);
    }
  }

}
```

Listing VI.17. *Funkcja USB_Receiver_Sender*

VI.6. Implementacja pomocnicza

Do implementacji pomocniczej należy klasa ułatwiająca wywoływanie aplikacji celem zebrania wyników. Prosta klasa posiadająca przeładowany operator dodawania do strumienia danych. W wypadku naszego projektu, dla normalnego działania aplikacji operator pozwala na wypisanie dowolnych danych, ale w wypadku włączenia zbierania danych w postaci kolumnowej ta klasa nie pozwoli na to.

```
class DebugPrinter
{
public:
  DebugPrinter() : _enabled(true)
  {
  }
  DebugPrinter(bool enabled) : _enabled(enabled)
  {
  }

  DebugPrinter& operator<<(const char ss[])
  {
    if(_enabled)
      std::cout << ss;
    return *this;
  }
  DebugPrinter& operator<<(const int& d)
  {
    if(_enabled)
      std::cout << d;
    return *this;
  }
  void set(bool enabled)
  {
    _enabled = enabled;
  }

private:
  bool _enabled;
};
```

Listing VI.18. *Klasa DebugPrinter*

W wypadku wywołania aplikacji:

```
use: libusb [S/A] <bufforSize> <fullDataSize> optional:<printOnlyResult>
```

Listing VI.19. *Uruchomienie testu*

bez ostatniego argumentu, w wypadku wykonania tego fragmentu kodu:

```
debugPrinter << "It is impossible to send " << fullDataSize << "B using " <<
    bufforSize << "B, to simplify application work sending " << (++count) * bufforSize
     << "B\n";
```

Listing VI.20. *Wypisywanie w zależności od argumentu wywołania aplikacji*

aplikacja wypisze powyższy tekst, natomiast w wypadku wywołania aplikacji wraz z ostatnim opcjonalnym argumentem <printOnlyResult>=1 spowoduje, że przeładowany operator klasy wspomnianej w listingu VI.18 zablokuje wypisywanie tekstu.
Klasa została stworzona tylko po to aby ułatwić przyszłe zbieranie wyników.

Poniżej (listing VI.21 został przedstawiony prosty makefile używany przy kompilacji pod system Unix.

```
libusb: Mode.o SynchMode.o AsynchMode.o Mode.hpp Mode.cpp SynchMode.hpp SynchMode.cpp
    AsynchMode.hpp
  g++ -std=c++11 libusbtest2.cpp SynchMode.cpp AsynchMode.cpp Mode.cpp -o libusb -I/
    usr/include/libusb-1.0 -lusb-1.0 -lpthread

AsynchMode.o: Mode.o Mode.hpp Mode.cpp AsynchMode.hpp AsynchMode.cpp
  g++ -std=c++11 -c AsynchMode.cpp -o AsynchMode.o -I/usr/include/libusb-1.0 -lusb-1.0
    -lpthread

SynchMode.o: Mode.o Mode.hpp Mode.cpp SynchMode.hpp SynchMode.cpp
  g++ -std=c++11 -c SynchMode.cpp -o SynchMode.o -I/usr/include/libusb-1.0 -lusb-1.0 -
    lpthread

Mode.o: Mode.hpp Mode.cpp
  g++ -std=c++11 -c Mode.cpp -o Mode.o -I/usr/include/libusb-1.0 -lusb-1.0 -lpthread

clean:
  @rm -f *.o libusb
```

Listing VI.21. *Makefile*

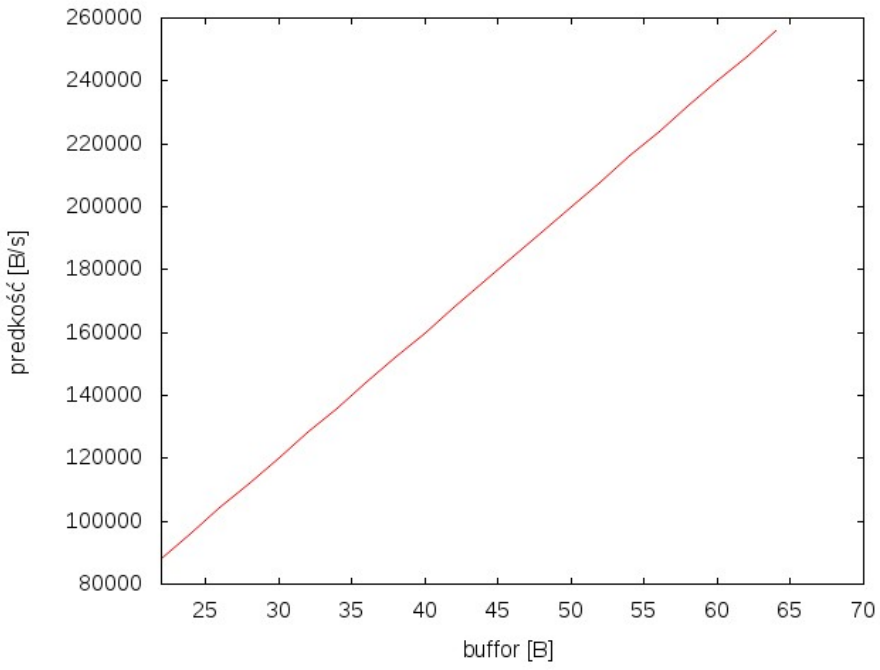
VII. Wyniki symulacji

Testy zostały przeprowadzone z użyciem prostej aplikacji w dwóch trybach: synchronicznym i asynchronicznym. Przedstawione poniżej wyniki wraz z krótką interpretacją przekazują zarys doświadczenia. Celem pracy było uzyskanie prędkości nie mniejszej niż 140 MBit/s co jest równe 17,5 MB/s.

Niestety okazało się iż mikrokontroler LandTiger pomimo wlutowanego gniazda USB2.0 nie jest w stanie obsłużyć tego standardu. Powodem tego jest to iż chip na płytce jest w stanie obsłużyć tylko standard USB1.1.

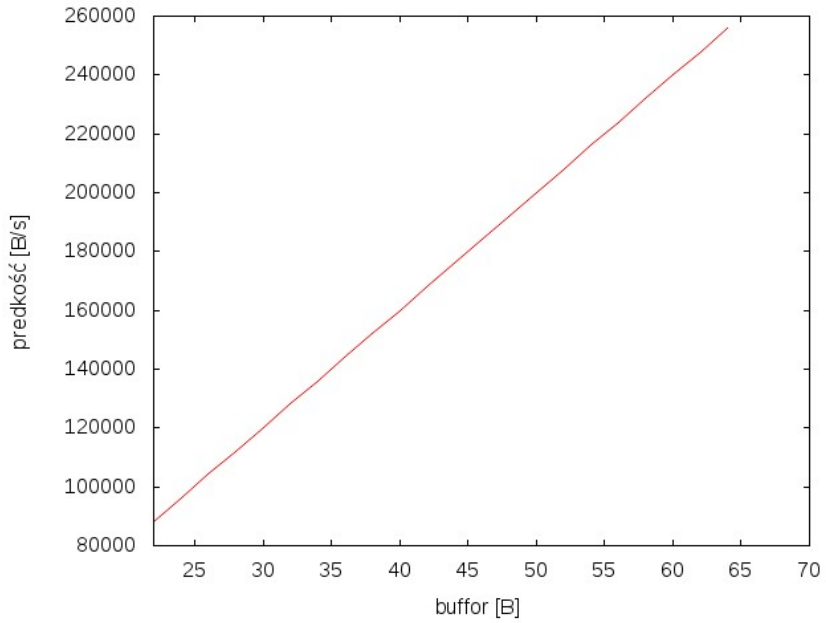
Powodem dla którego mikrokontroler działa pomimo obsługi USB1.1 (a nie jak wspomniano USB2.0) jest to że wlutowane gniazdo ustandaryzowane jako USB2.0 jest wstecznie kompatybilne (więcej informacji w rozdziale II.2).

Zebrane wyniki możemy skategoryzować według: ilości przesłanych danych, szybkości przesyłu danych z komputera osobistego do kontrolera, szybkości przesyłu danych z kontrolera do komputera osobistego, szybkości przesyłu w obie strony. Kluczowym elementem jest tutaj wielkość buffora, czyli ilości danych możliwych do wysłania/odebrania w jednym tiku zegara. Niestety w wypadku USB1.1 jest to 64B (z powodu ograniczeń na chipie płytki). Większość wyników została wygenerowana za pomocą programu który nie pozwalał na przekroczenie dopuszczalnego przez LandTiger'a bufforu (tak jak w przypadku Rys. VII.1 oraz innych)



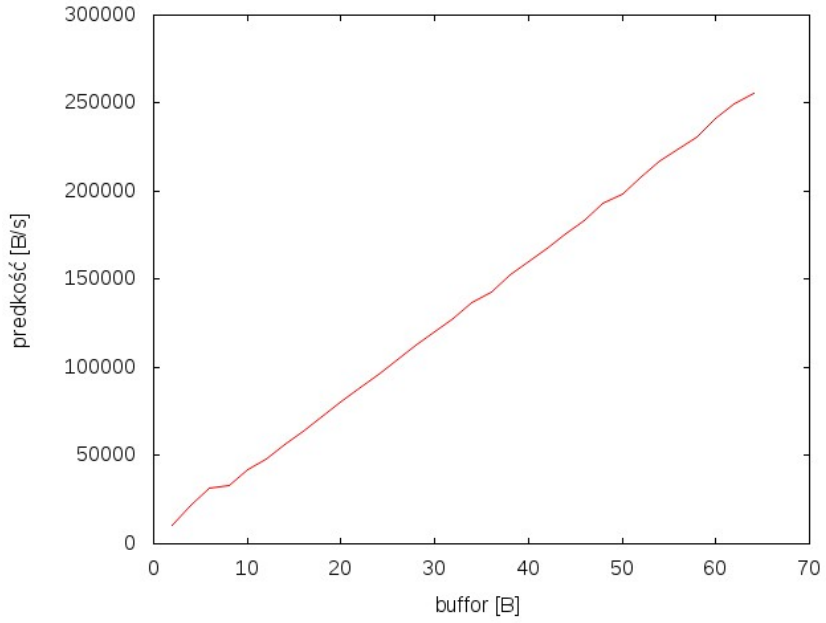
Rys. VII.1. *Wykres ilustrujący prędkość wysyłania danych w zależności od buffora dla 102 MB (tryb synchroniczny)*

Na Rys. VII.1 widoczny jest wyraźny wzrost szybkości wysyłania danych wraz ze zwiększeniem bufora. Jest to doświadczenie wykonane na stosunkowo małym buforze spowodowane ograniczeniami płytki LandTiger. Wykres jest liniowy



Rys. VII.2. *Wykres ilustrujący prędkość odbierania danych w zależności od buffora dla 102 MB (tryb synchroniczny)*

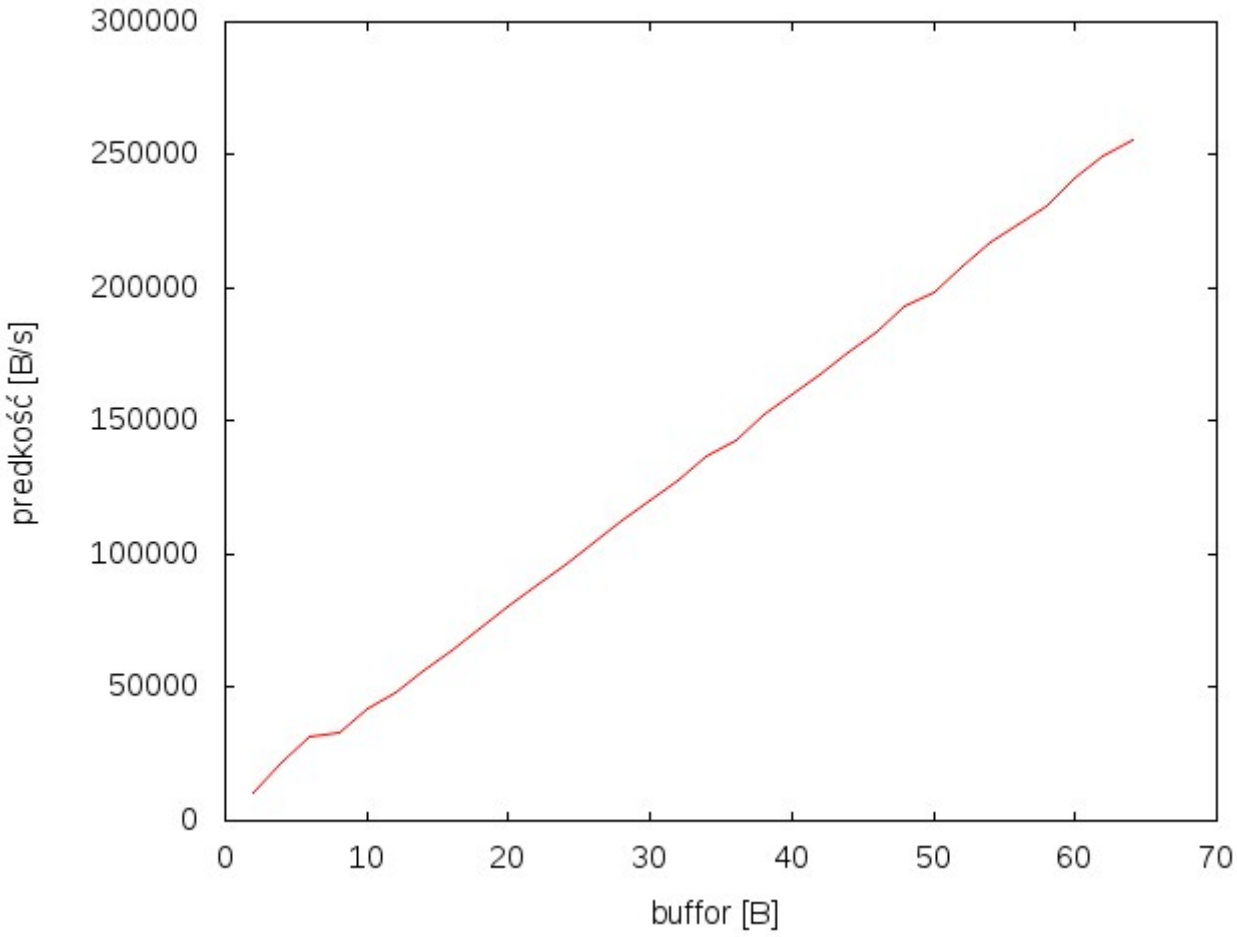
Analogiczny wykres widoczny jest dla prędkości odbierania danych (Rys. VII.2), różnice są niezauważalne przy tak małej ilości danych.



Rys. VII.3. *Wykres ilustrujący prędkość wysłania danych w zależności od buffora dla 10 MB (tryb synchroniczny)*

W wypadku wysyłania nieco mniejszej próbki danych prędkość obrazuje się jako nieco

mniej stabilna. Całość widoczna jest na Rys. VII.3. Wykres nie jest już liniowy a prędkość zdaje się być na bardziej niedeterministyczna.

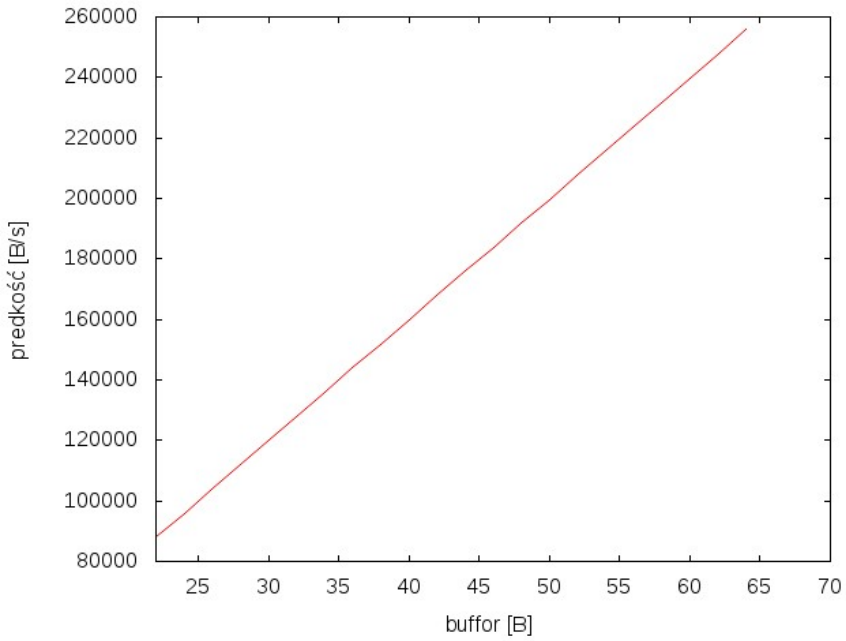


Rys. VII.4. *Wykres ilustrujący prędkość odbierania danych w zależności od buffora dla 10 MB (tryb synchroniczny)*

Podobnie jak w poprzednim przypadku (dla większej ilości danych), wykres prędkości odbierania danych pokrywa się z wykresem prędkości wysyłania danych (Rys. VII.4).

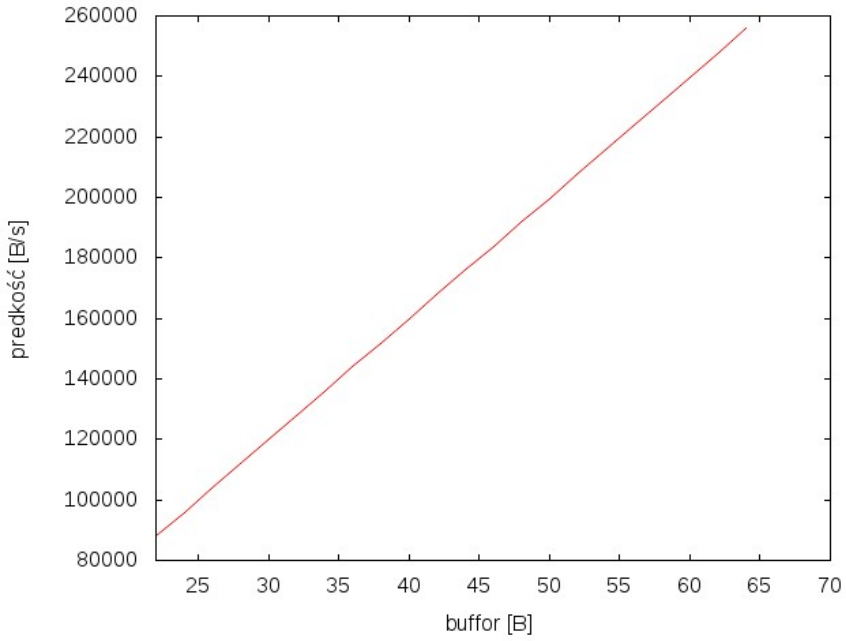
Podsumowując synchroniczne wysyłanie danych nie należy do najbardziej efektywnych metod. Każda operacja zwraca status i ten status musi być weryfikowany aby zlokalizować ewentualny błąd.

Zupełnie innym podejściem charakteryzuje się metoda Asynchroniczna w której korzysta się z nie blokujących metod. Tutaj inną przeszkodą jest to aby to programista zadbał o synchronizacje (ponieważ po łączu w jednym czasie mogą przebiegać dane tylko w jedną stronę). Dzięki tej metodzie istnieje możliwość uzyskania lepszych (szybszych) wyników.



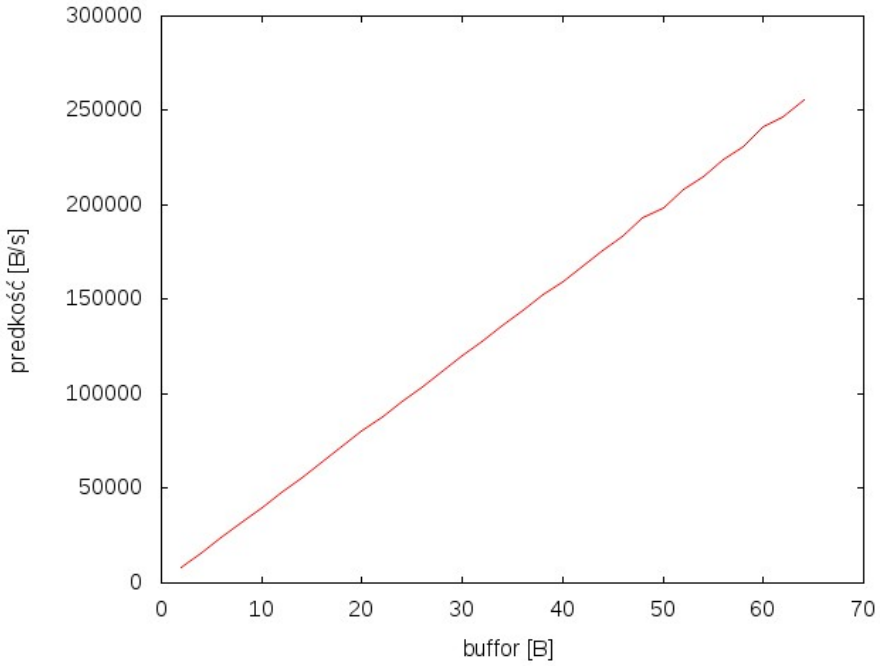
Rys. VII.5. *Wykres ilustrujący prędkość wysyłania danych w zależności od buffora dla 102 MB (tryb Asynchroniczny)*

Jak widac na Rys. VII.5 zależność liniowa również występuje dla tej ilości danych orzy przesyle asynchronicznym, podobnie jak w przypadku przesyłu synchronicznego (Rys. VII.1).



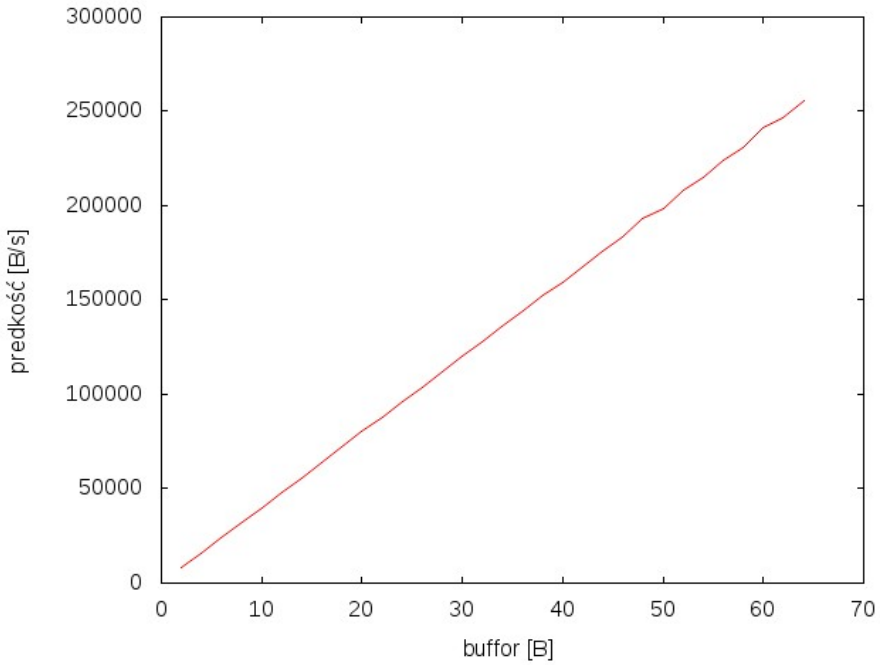
Rys. VII.6. *Wykres ilustrujący prędkość odbierania danych w zależności od buffora dla 102 MB (tryb Asynchroniczny*

Analogiczna sytuacja występuje w wypadku odbierania danych (o wielkości 102 MB) metodą asynchroniczną. Wykres jest liniowy jak jego odpowiednik z metody synchronicznej (Rys. VII.2) a wyniki są bardzo zbliżone do siebie.



Rys. VII.7. *Wykres ilustrujący prędkość wysyłania danych w zależności od buffora dla 10 MB (tryb asynchroniczny)*

Jeśli przeprowadzimy testy dla mniejszej ilości danych to zauważalny jest brak liniowości takiego wykresu (zarówno dla Rys. VII.7 jak i Rys. VII.8).

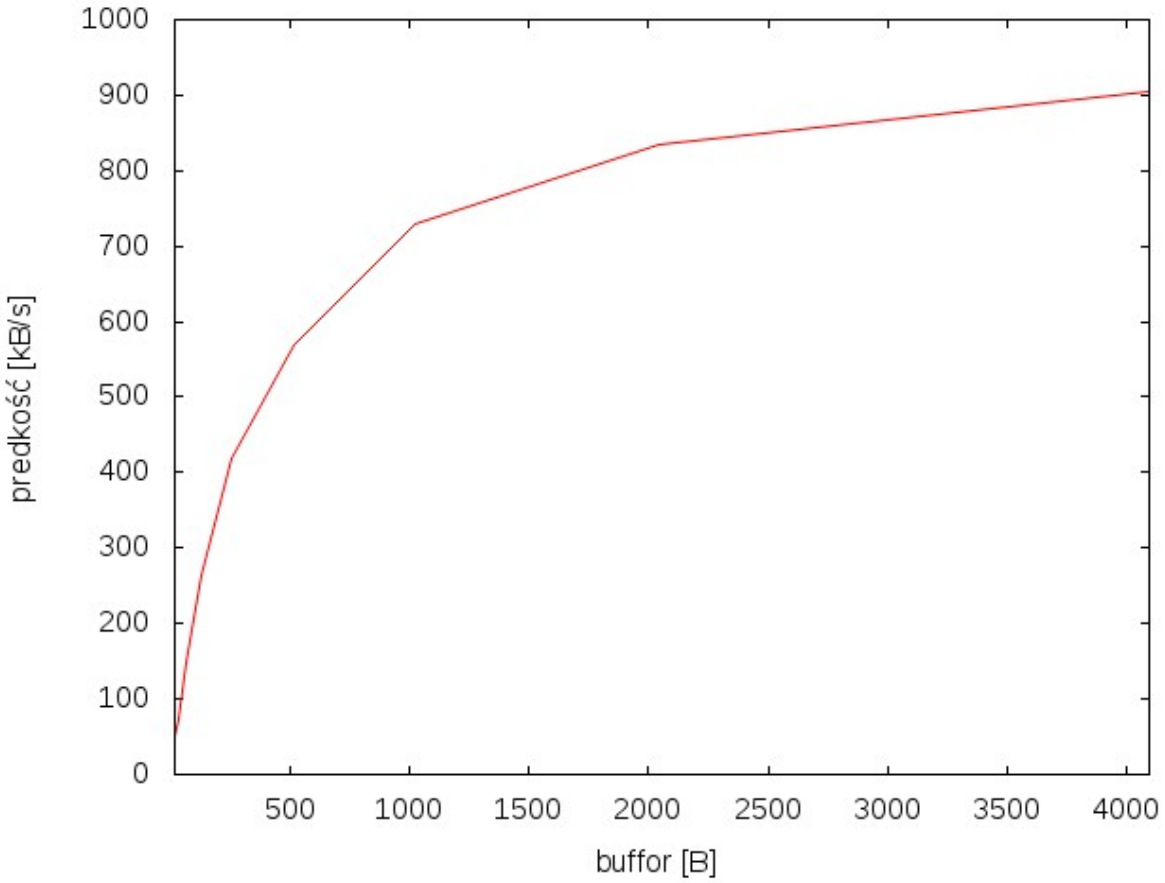


Rys. VII.8. *Wykres ilustrujący prędkość odbierania danych w zależności od buffora dla 10 MB (tryb asynchroniczny)*

Podsumowując użycie bufora dopuszczalnego przez mikrokontroler LandTiger (czyli bufora nie większego niż 64B) nie jest nawet zbliżona do prędkości porządanej (140 MB/s)

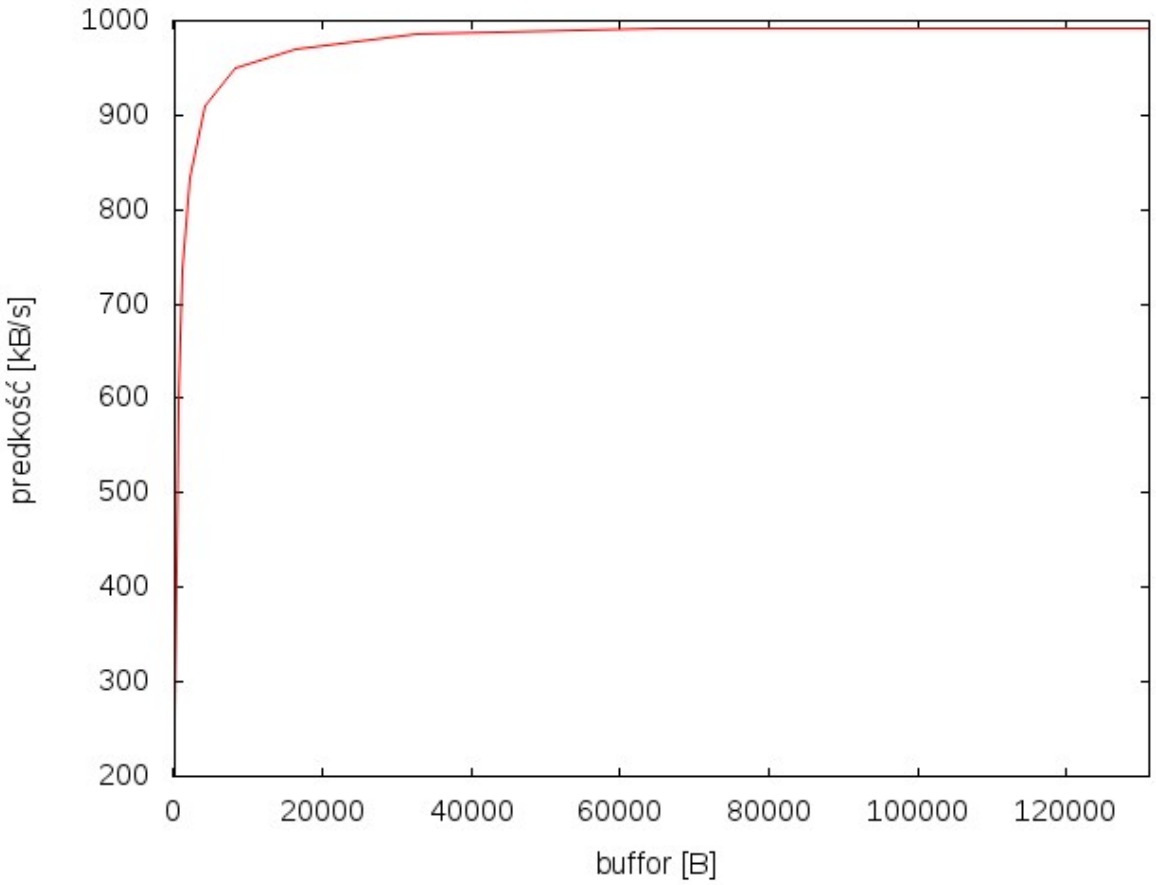
bez względu czy korzystaliśmy z synchronicznej metody czy asynchronicznej. Przyczyną jest tutaj wielkość używanego bufora, z powodów dla których LandTiger nie dopuszcza użycia wiekszego bufora niż 64B pozostałem testy oparte są na zasymulowanych danych.

Dodatkowo została przeprowadzona symulacja z uwzględnieniem większego bufora danych ale została wykonana bez poprawnego odbioru danych po stronie kontrolera. Oznacza to że te dane mogą zostać potraktowane jedynie jako przypuszczenia jak wyglądałby wykres gdyby dane były przetworzone prawidłowo. Poniższe (Rys. VII.9 oraz Rys. VII.10 wykresy przedstawiają wygenerowane wyniki.



Rys. VII.9. *Wykres ilustrujący zmianę prędkości przesyłania danych z użyciem bufora z przedziału 16 B - 4 kB*

Powyższy wykres obrazuje zmianę prędkości wysyłania przy założeniu poprawnego odbioru danych po stronie kontrolera. Zależność liniowa całkowicie zaniknęła. Wartości rozmiaru bufora danych jest podwajana z każdym zgięciem krzywej (początkowa wartość 16 B, końcowa 4096 B = 4 kB), a więc rozbieżność przesyłu danych jest duża. Warto zauważyć iż pomimo dość dużego bufora danych prędkość nie przekroczyła 1 MB/s (max wartość 905,702 kB/s).



Rys. VII.10. *todo:S_bbuf2*

Powyższy wykres jest rozwinięciem poprzednika. Obrazuje zmianę prędkości na podstawie zasymulowanych danych oraz przy założeniach poprawnego odebrania danych przez mikrokontroler. Podstawową różnicą jest wielkość użytego bufora w tym przypadku. Dla tego przypadku został użyty bufor o wartości minimalnej równe 128 B oraz maksymalnej równej 131072 B (128 kB). Bufor w każdym kroku jest podwajany. Początkowo wartość uzyskiwanej prędkości wzrasta gwałtownie w każdym kroku, natomiast powyżej wartości 8 kB nie są już tak gwałtowne a wręcz krzywa przechodzi w coraz bardziej poziomą i dąży do wartości 1 MB/s.

Pomimo zasymulowania dość sporych danych wraz z użyciem dość dużego bufora danych nie udało się uzyskać oczekiwanej minimalnej prędkości. Jednym z powodów jest to, że obsługa odbioru danych została zasymulowana po stronie mikrokontrolera dla bufora danych większego niż 64 B (ograniczenia kontrolera).

VIII. Podsumowanie

Celem projektu było uzyskanie jak największej prędkości przesyłu danych po interfejsie USB. Na wstępie zamieszczone zostało krótkie wprowadzenie odnośnie dostępnych standardów USB oraz ich zarysowi historycznemu. Kolejnym krokiem był dwóch najbardziej popularnych bibliotek do obsługi interfejsu USB (libUSB oraz winUSB). W rozdziale został zawarty krótki opis najważniejszych aspektów bibliotek wraz z uzasadnieniem wyboru libUSB jako tej używanej w projekcie.

Dosyć ważną część odgrywa opis API biblioteki libUsb. Znajduje się w nim dość obszerny opis poszczególnych funkcji użytych w projekcie. Jest to kluczowe aby zrozumieć późniejszą implementację.

W kolejnym rozdziale został przedstawiony mikrokontroler użyty w projekcie, wraz z dokładnym opisem dostępnych funkcjonalności (nie tylko tych użytych w projekcie). Jak okazuję się w kolejnych rozdziałach brakuje mu kluczowej funkcjonalności z punktu widzenia projektu a mianowicie obsługi standardu USB2.0. [14]

Jednym z najważniejszych rozdziałów jest rozdział opisujący sposób implementacji projektu wraz z załączonym kodem źródłowym. Przedstawiony został tam sposób łatwego wywoływania testu oraz opis krok po kroku co zostaje wykonane przed uzyskaniem rezultatu. Zostaje też przedstawiony łatwy sposób rozszerzalności o kolejne klasy implementujące nowy rodzaj testu.

Rozdział obrazujący rezultaty testów jest najważniejszym w projekcie. Wnioski z wyników nie należą do pozytywnych. Z powodu ograniczeń na mikrokontrolerze w tym ograniczenie obsługi na chipie do standardu USB1.1 (pomimo wlutowanego złącza USB2.0) nie udało się uzyskać minimalnej wymaganej w projekcie prędkości przesyłania danych (17,5 MB/s). Nawet pomimo zasymulowania większego bufora danych (bez poprawnej obsługi odbioru danych po stronie mikrokontrolera w którym zaniechano weryfikacji odebranych danych z powodu niemożności obsługi tak dużego buforu danych).

Warto zaznaczyć, że nie oznacza to iż uzyskanie docelowej prędkości z wykorzystaniem standardu USB2.0 jest nie możliwe (oczywiście z wykorzystaniem innego mikrokontrolera). Standard USB2.0 dopuszcza wielkość bufora 1024 B a co za tym idzie porównując to do obecnej wielkości uzyskamy 16 razy większą.

Na podstawie otrzymanych wyników uzyskanych z wykorzystaniem standardu USB1.1 możemy założyć iż prędkość przesyłu danych z wykorzystaniem USB2.0 oscylowała by w granicach 16 MB/s. Wartość ta jest zależna od odpowiedniej konfiguracji więc możemy dodatkowo przyjąć prędkość 17,5 MB/s jest jak najbardziej osiągalna.

Bibliografia

[1] Michael J. Pont *Embedded C* , UK: Pearson Education Limited, 2002.

[2] Steven F. Barret and Daniel J. Pack *Embedded Systems* USA: Pearson Education, Inc., 2005.

[3] Don Anderson *Universal Serial Bus System Architecture* USA: MindShare, Inc., 1997.

[4] Rajaram Regupathy *Bootstrap Yourself with Linux-USB Stack: Design, Develop, Debug, and Validate Embedded USB* Course Technology, 2012.

[5] Wooi Ming Tan *Developing USB PC Peripherals* Annabooks, 1999.

[6] USB Implementers Forum, Inc., USB2.0 specification reference [online]
`http://www.usb.org/developers/docs/usb20_docs/`

[7] USB Implementers Forum, Inc., USB3.0 and USB2.0 specyfication reference [online]
`http://www.usb.org/developers/docs/`

[8] libusb 2012-2015, libusb documentation reference [online]
`http://libusb.sourceforge.net/api-1.0/`

[9] libusb 2012-2015, libusb description reference [online]
`http://libusb.info/`

[10] Microsoft 2015, winusb description reference [online]
`https://msdn.microsoft.com/en-us/library/windows/hardware/ff540196(v=vs.85).aspx`

[11] Microsoft 2015, *Developing Windows applications for USB devices* [online]
`https://msdn.microsoft.com/en-us/library/windows/hardware/dn303342(v=vs.85).aspx`

[12] Microsoft 2015, *How to Access a USB Device by Using WinUSB Functions* [online]
`https://msdn.microsoft.com/en-us/library/windows/hardware/ff540174(v=vs.85).aspx`

[13] Microsoft 2010, *How to Use WinUSB to Communicate with a USB Device* [doc from Microsoft web]
`http://download.microsoft.com/download/9/C/5/9C5B2167-8017-4BAE-9FDED599BAC8184A WinUsb_HowTo.docx`

[14] mbed, LandTiger description reference [online]
`https://developer.mbed.org/users/wim/notebook/landtiger-baseboard/`